计算机科学与技术专业课程

计算机组成

MIPS单周期数据通路

高小鹏

北京航空航天大学计算机学院
系统结构研究所

提纲

- 内容主要取材
  - CS617的20讲
- 处理器设计
- 数据通路概述
- 组装数据通路
- 控制介绍

Great Idea #1: Levels of Representation/Interpretation

Higher-Level Language Program (e.g. C)

Compiler

Assembly Language Program (e.g. MIPS)

Assembler

Machine Language Program (MIPS)

Machine Interpretation

Hardware Architecture Description (e.g. block diagrams)

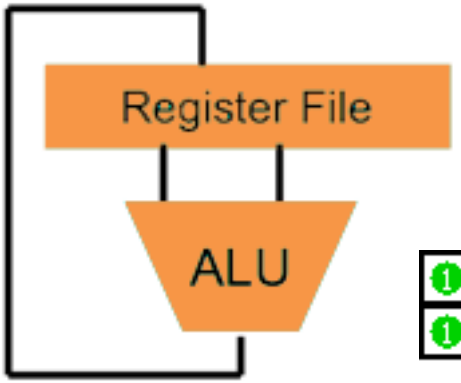
Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

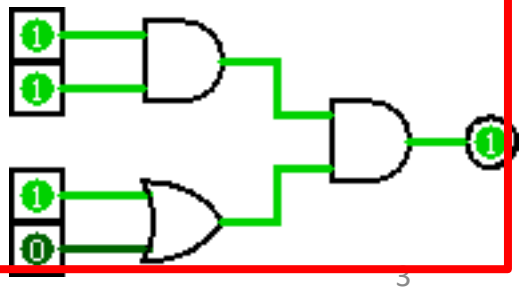
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)
```

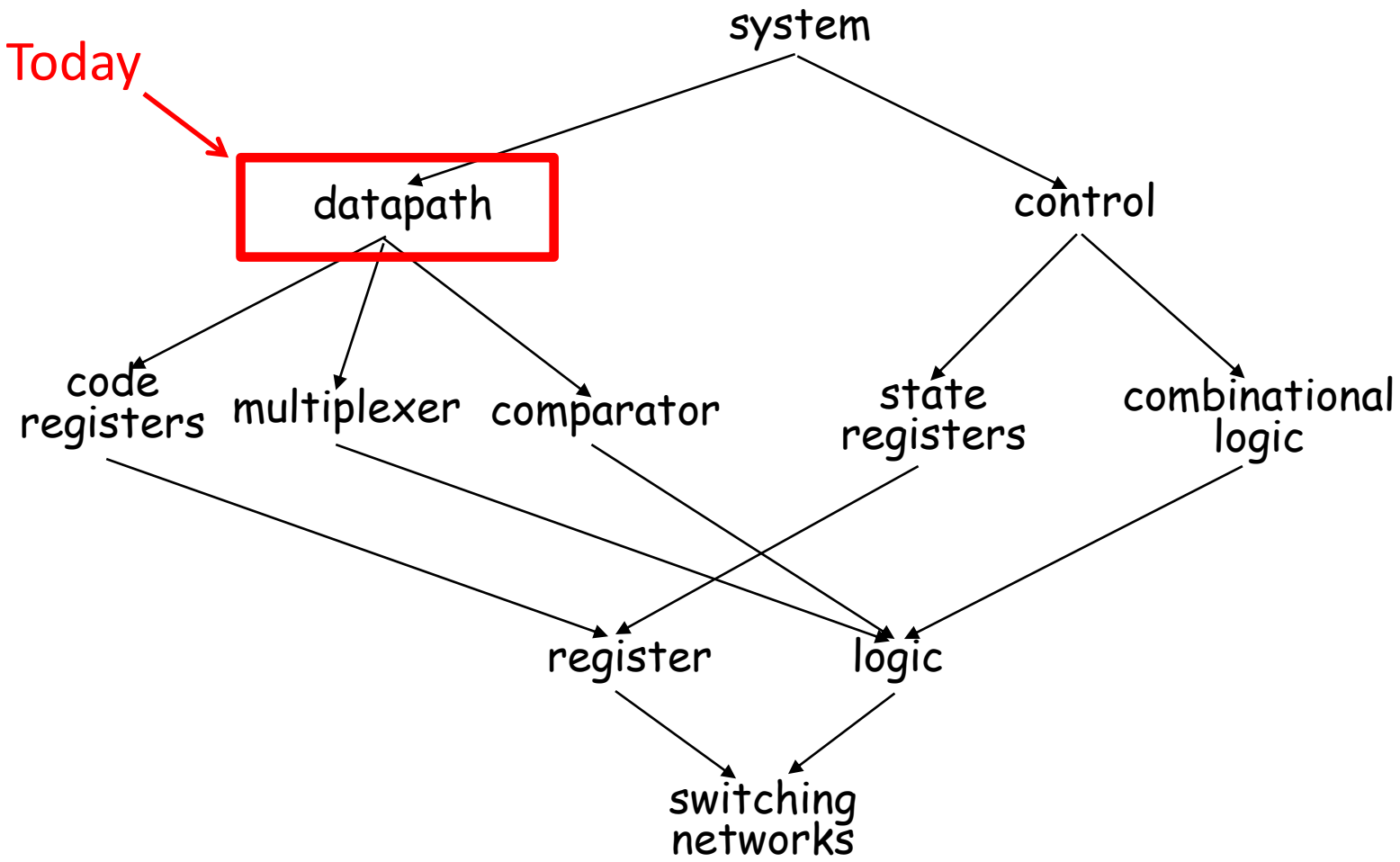
```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```



We are here



Hardware Design Hierarchy

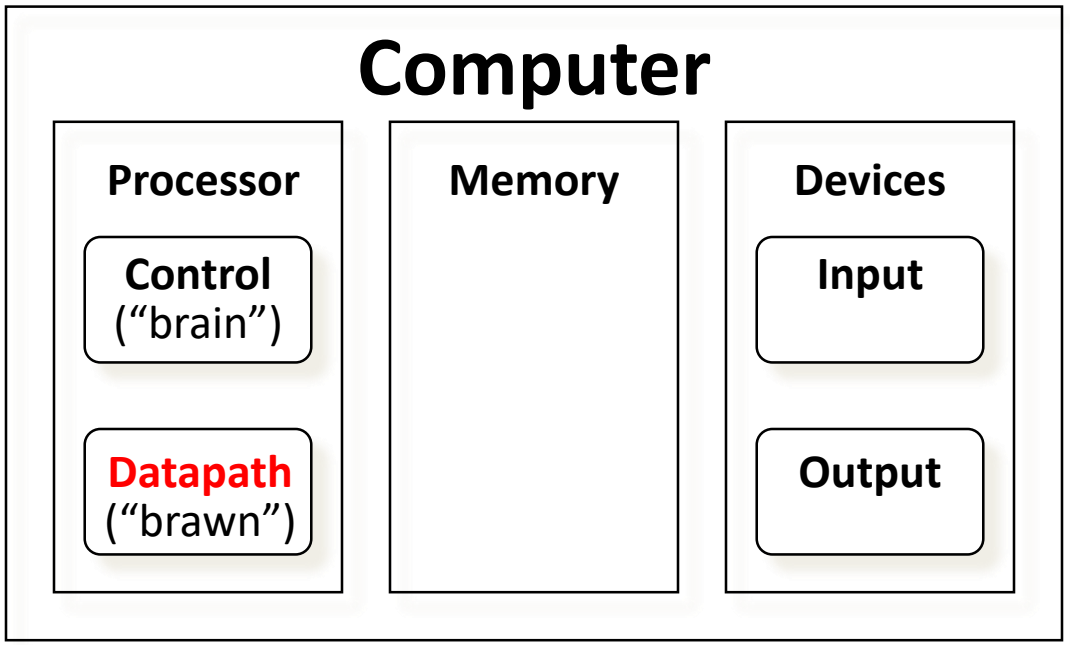


提纲

- 内容主要取材
  - CS617的20讲
- 处理器设计
- 数据通路概述
- 组装数据通路
- 控制介绍

Five Components of a Computer

- Components a computer needs to work
  - Control
  - Datapath
  - Memory
  - Input
  - Output



The Processor

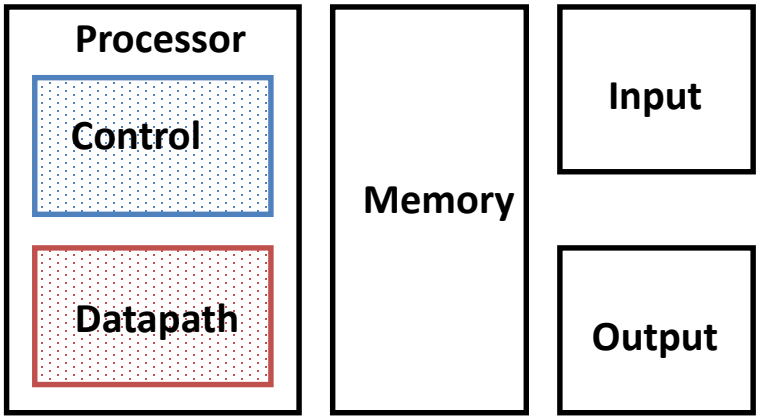
- ***Processor (CPU):*** Implements the instructions of the Instruction Set Architecture (ISA)
  - *Datapath:* part of the processor that contains the hardware necessary to perform operations required by the processor ("the brawn")
  - *Control:* part of the processor (also in hardware) which tells the datapath what needs to be done ("the brain")

Processor Design Process

- Five steps to design a processor:

Now

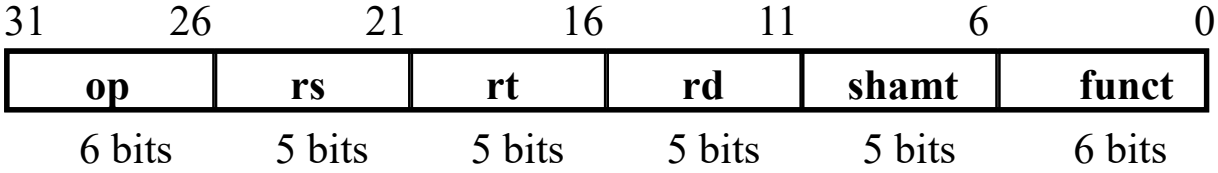
1. Analyze instruction set → datapath requirements
2. Select set of datapath components & establish clock methodology
3. Assemble datapath meeting the requirements



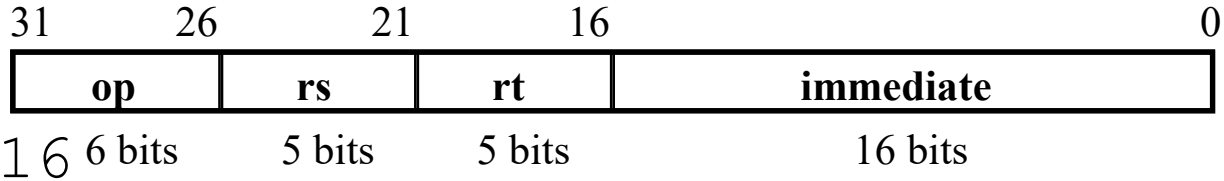
4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer
5. Assemble the control logic
   - Formulate Logic Equations
   - Design Circuits

The MIPS-lite Instruction Subset

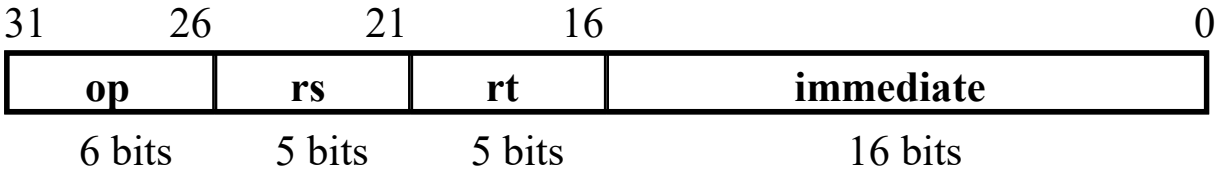
- ADDU and SUBU
  - `addu rd,rs,rt`
  - `subu rd,rs,rt`



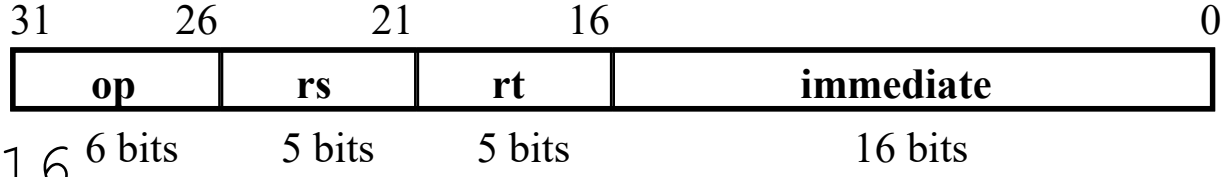
- OR Immediate:
  - `ori rt,rs,imm16`



- LOAD and STORE Word
  - `lw rt,rs,imm16`
  - `sw rt,rs,imm16`



- BRANCH:
  - `beq rs,rt,imm16`



Register Transfer Language (RTL)

- All start by *fetching* the instruction:

```
R-format:  {op, rs, rt, rd, shamt, funct} ← MEM[ PC ]

I-format:  {op, rs, rt, imm16} ← MEM[ PC ]
```

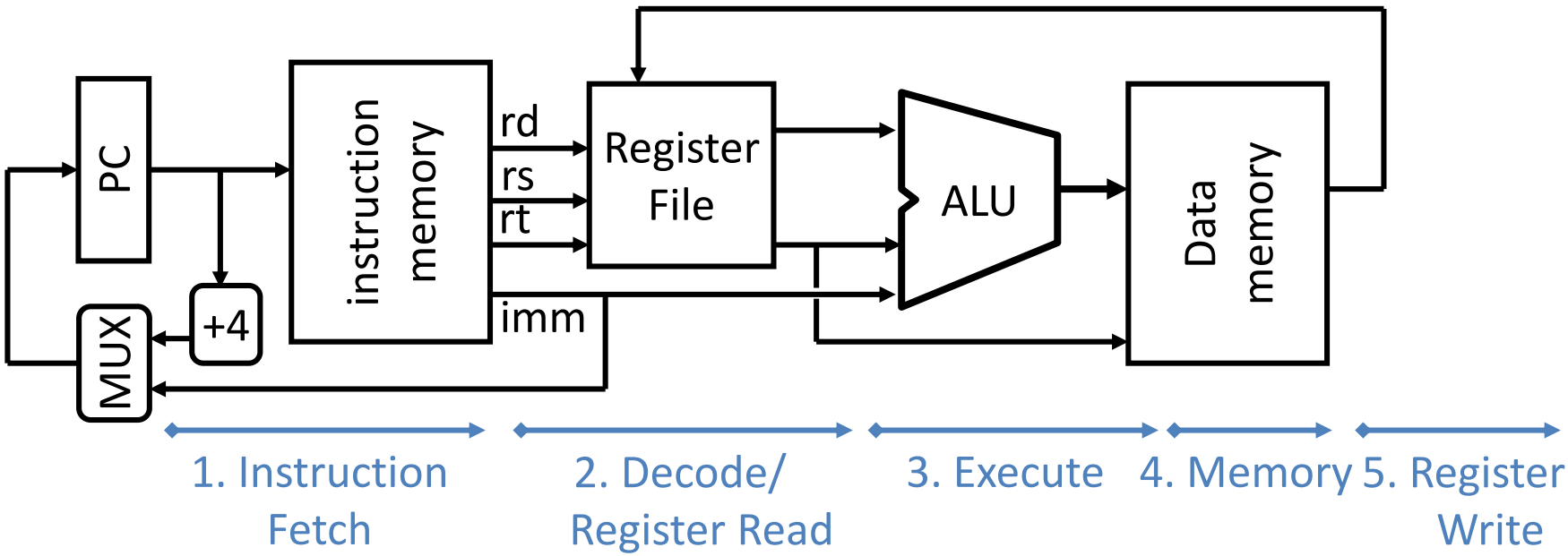
- RTL gives the meaning of the instructions:

```
Inst   Register Transfers

ADDU   R[rd]←R[rs]+R[rt]; PC←PC+4

SUBU   R[rd]←R[rs]-R[rt]; PC←PC+4

ORI    R[rt]←R[rs]|zero_ext(imm16); PC←PC+4

LOAD   R[rt]←MEM[R[rs]+sign_ext(imm16)]; PC←PC+4

STORE  MEM[R[rs]+sign_ext(imm16)]←R[rt]; PC←PC+4

BEQ    if ( R[rs] == R[rt] )
           then PC←PC+4 + (sign_ext(imm16) || 00)
           else PC←PC+4
```

Step 1: Requirements of the Instruction Set

- Memory (MEM)
  - Instructions & data (separate: in reality just caches)
  - Load from and store to
- Registers (32 32-bit regs)
  - Read *rs* and *rt*
  - Write *rt* or *rd*
- PC
  - Add 4 (+ maybe extended immediate)
- Extender (sign/zero extend)
- Add/Sub/OR unit for operation on register(s) or extended immediate
  - Compare if registers equal?

Generic Datapath Layout

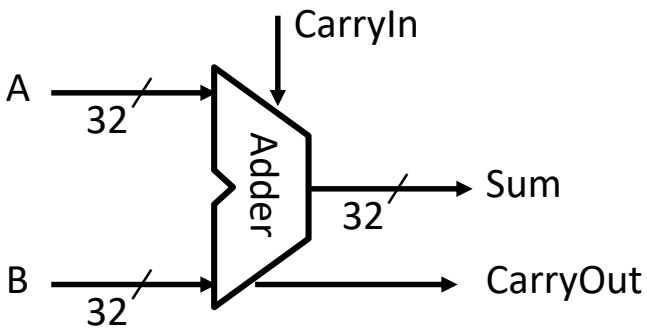


- Break up the process of "executing an instruction"
  - Smaller phases easier to design and modify independently
- ~~Proj1 had 3 phases: Fetch, Decode, Execute~~
  - ~~Now expand Execute~~

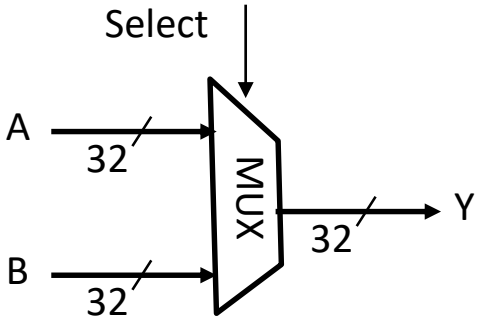
register file~寄存器堆/寄存器文件 decode~译码/解码

Step 2: Components of the Datapath

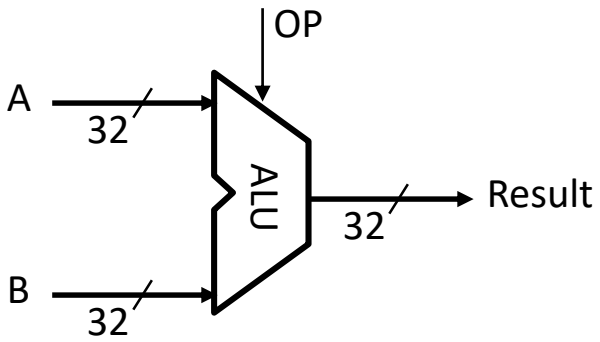
- Combinational Elements
  - Gates and wires
- State Elements + Clock
- Building Blocks:



Adder



Multiplexer



ALU

MUX设计

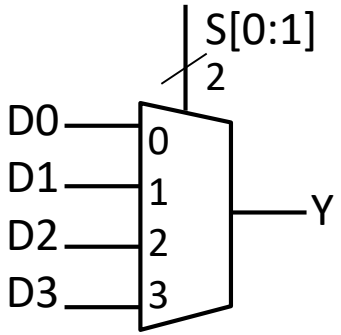
- 1位2选1 MUX表达式：**Y = !S&D0 + S&D1**
- 1位4选1 MUX表达式：**Y = ?**

```
Y = !S0&!S1&D0 +
    !S0& S1&D1 +
     S0&!S1&D2 +
     S0& S1&D3
```

- 1位N选1 MUX表达式？

$$\mathbf{Y} = \sum_{\mathbf{0}}^{\boldsymbol{N-1}} \boldsymbol{S_i \& D_i}$$

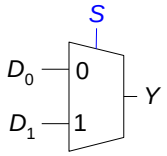
Si：S的组合表示



Multiplexer (Mux)

- Selects between one of *N* inputs to connect to output
- $\log_2 N$-bit select input – control input
- **Example:**

2:1 Mux



S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

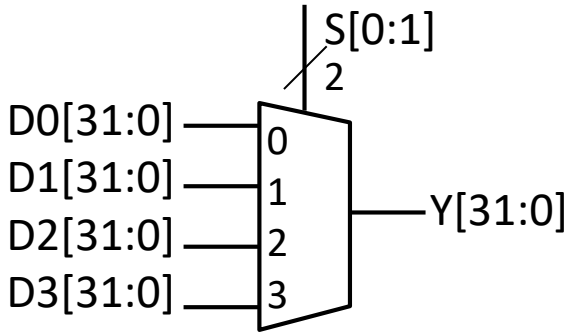
S	Y
0	D_0
1	D_1

MUX设计

- 32位4选1 MUX表达式：？
  - 就是32个4选1 MUX
  - Verilog表达式

```
assign Y = !S0&!S1&D0 +
           !S0& S1&D1 +
            S0&!S1&D2 +
            S0& S1&D3 ;
```

- M位N选1 MUX表达式？



TIP

表达式简写表达方法，即省略了[31:0]

前提：D0～D3及Y的位宽一致！

ALU Requirements

- MIPS-lite: add, sub, OR, equality

```
ADDU    R[rd] = R[rs] + R[rt]; ...
SUBU    R[rd] = R[rs] - R[rt]; ...
ORI     R[rt] = R[rs] | zero_ext(Imm16)...
BEQ     if ( R[rs] == R[rt] )...
```

- Equality test: Use subtraction and implement output to indicate if result is 0
- P&H also adds AND, Set Less Than (1 if A < B, 0 otherwise)
- Can see ALU from P&H C.5 (on CD)

VerilogHDL建模4位加法与减法

- 4位加法（无overflow检测）

4位加法

```
1  module add4( a, b, c ) ;
2      input  [3:0]  a, b ;
3      output [3:0]  c ;
4
5      assign c = a + b ;
6
7  endmodule
```

TIP：
Verilog的“+”：最基础的二进制加法，是不区分正负数的！

- 4位减法（采用二进制补码运算方法，即加相反数）

4位减法（无overflow检测）

```
1  module sub4( a, b, c ) ;
2      input  [3:0]  a, b ;
3      output [3:0]  c ;
4
5      assign c = a + ~b + 1'b1 ;
6
7  endmodule
```

TIP：
也可以采用
assign c = a - b ;

VerilogHDL建模4位ALU

- ALU的功能：加、减、或
  - 加/减：不支持overflow
- op：控制信号
  - 00~加法；01~减法；10~或；11~保留

head.v

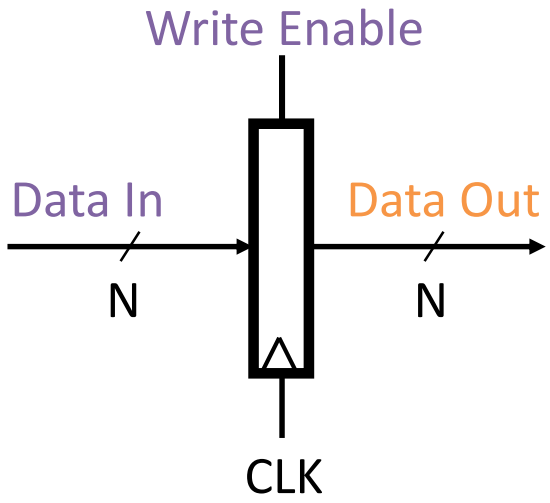
```
`define ALU_ADDU 2'b00
`define ALU_SUBU 2'b01
`define ALU_OR   2'b10
```

- 建模方法：利用assign语句实现加法与减法的2选1

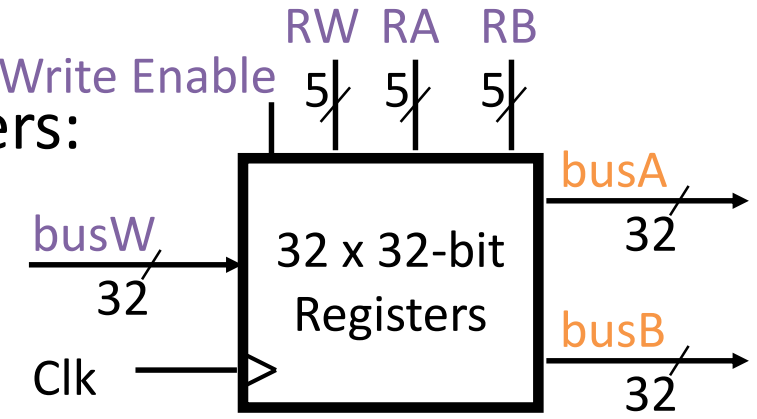
```
1  `include "head.v"
2
3  module ALU( a, b, c, op ) ;
4      input  [3:0]  a, b ;
5      input  [1:0]  op ;
6      output [3:0]  c ;
7
8      assign c = (op==`ALU_ADDU) ? (a + b)       :   加
9                 (op==`ALU_SUBU) ? (a + ~b + 1)  :   减
10                (op==`ALU_OR)   ? (a | b)       :   或
11                                  4'b0000 ;       保留
12 endmodule
```

Storage Element: Register

- As seen in Logisim intro
  - N-bit input and output buses
  - Write Enable input
- Write Enable:
  - De-asserted (0): Data Out will not change
  - Asserted (1): Data In value placed onto Data Out on the rising edge of CLK



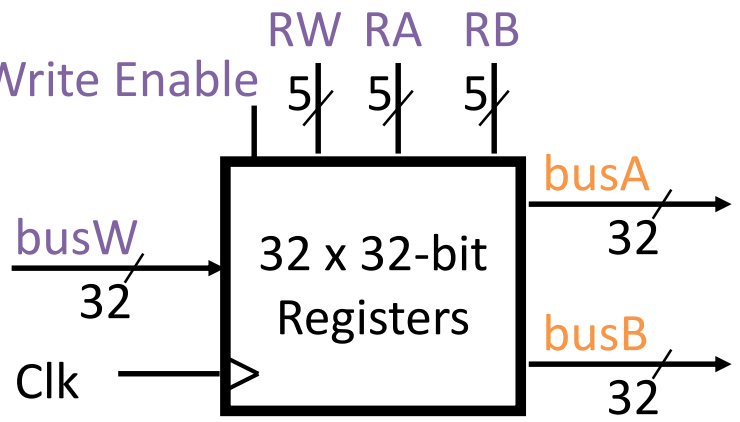
Storage Element: Register File

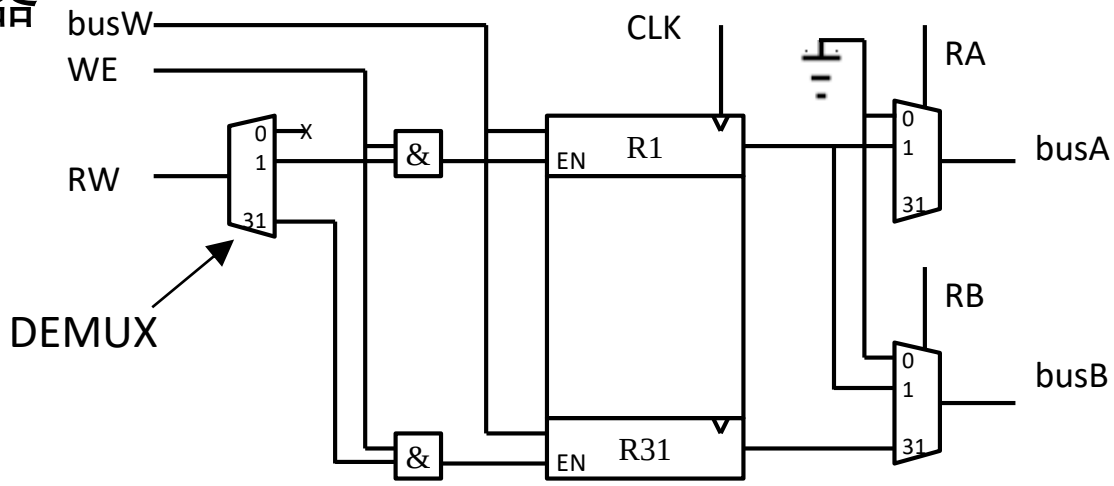


- *Register File* consists of 32 registers:
  - Output buses busA and busB
  - Input bus busW
- Register selection
  - Place data of register RA (number) onto busA
  - Place data of register RB (number) onto busB
  - Store data on busW into register RW (number) when Write Enable is 1
- Clock input (CLK)
  - CLK input is a factor ONLY during write operation
  - During read, behaves as a combinational logic block: RA or RB valid → busA or busB valid after "access time"

Register File设计考虑

- 内部需要多少个32位寄存器？
- 读出数据功能：32位31选1 MUX
  - 2个读出端口是独立工作
  - 无需彼此等待
- 写入数据功能：关键是写使能
  - 每个寄存器需要一个写使能
- DEMUX：分离器/解码器
  - Demultiplexer
  - N位编码产生2^N个输出
  - 有且仅有1个输出有效





VerilogHDL建模寄存器

- 1位D寄存器：标准的寄存器

1位D寄存器

```
1  module d_ff( d, q, clk ) ;
2      input  d, clk ;
3      output q ;
4
5      reg    r ;
6
7      assign q = r ;
8      always @(posedge clk)
9          r <= d;
10
11 endmodule
```

r：寄存器

将r从q输出
告诉编译器，以下行为按寄存器建模
时钟上升沿时，将输入保存至r

VerilogHDL建模寄存器

- 1位写使能D寄存器

1位写使能D寄存器

```
 1  module d_ff( d, we, q, clk ) ;
 2       input  d, we ;
 3       output q ;
 4       input  clk ;
 5
 6       reg    r ;
 7
 8       assign q = r ;
 9       always @( posedge clk )
10           if ( we )
11                r <= d;
12
13  endmodule
```

寄存器建模的完整写法：

```
if ( we )
     r <= d ;
else
     r <= r ;
```

Q：为什么可以忽略else？

A：对于缺少的分支，编译器会自动补充“**r <= r**”。

VerilogHDL建模寄存器

- 1位写使能D异步复位寄存器

1位写使能D异步复位寄存器

```
1  module d_ff( d, we, q, clk, rst ) ;
2       input  d, we ;
3       output q ;
4       input  clk, rst ;
5
6       reg    r ;
7
8       assign q = r ;
9       always @( posedge clk or posedge rst )
10          if ( rst )
11              r <= 1'b0 ;
12          else if ( we )
13              r <= d ;
14
15 endmodule
```

TIP：分析方法

由于rst在敏感表中，因此rst的有效和clk的有效，均会导致always语句块执行。这意味着rst对d的作用与clk无关，故这就是异步复位。

Q：在设计中如何选择异步复位or同步复位？

VerilogHDL建模寄存器

- 32位写使能寄存器

32位写使能寄存器

```
1  module d32( d, we, q, clk ) ;
2       input  [31:0] d ;
3       input         we ;
4       output [31:0] q ;
5       input  clk ;
6
7       reg    [31:0] r ;
8
9       assign q = r ;
10      always @( posedge clk )
11          if ( we )
12              r <= d ;
13
14 endmodule
```

TIP：
对于N位寄存器，仅仅改变输入信号、输出信号和内部寄存器定义的位数即可。

VerilogHDL建模寄存器

- 32位写使能寄存器（用generate-for建模。Verilog-2001标准）

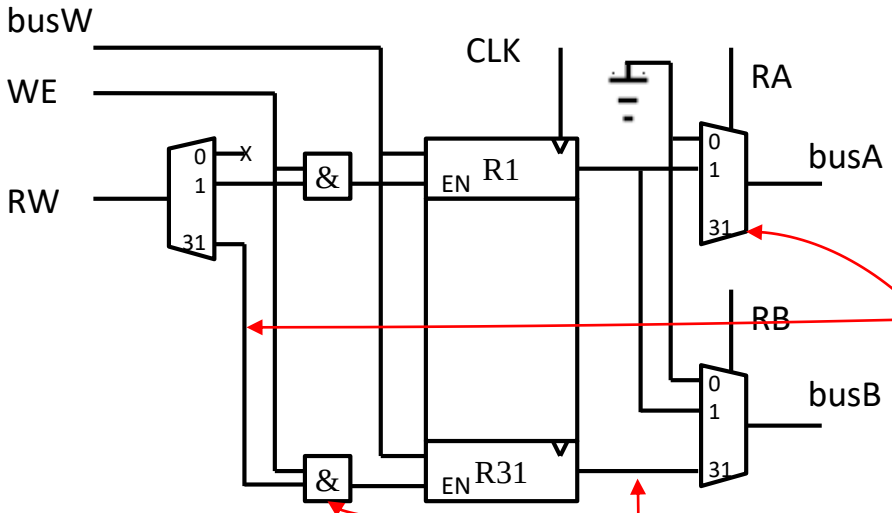
32位写使能寄存器

```
1  module d32( d, we, q, clk ) ;
2       input  [31:0] d ;
3       input         we ;
4       output [31:0] q ;
5       input  clk ;
6
7       genvar        i ;
8
9       generate
10          for ( i=0; i<32; i=i+1 )
11          begin : label_d
12              d_ff u_dff(d[i], we, q[i], clk) ;
13          end
14      endgenerate
15
16 endmodule
```

TIP
1）genvar定义循环变量
2）必须要有for、begin/end
3）begin必须要有标签

Verilog建模RF

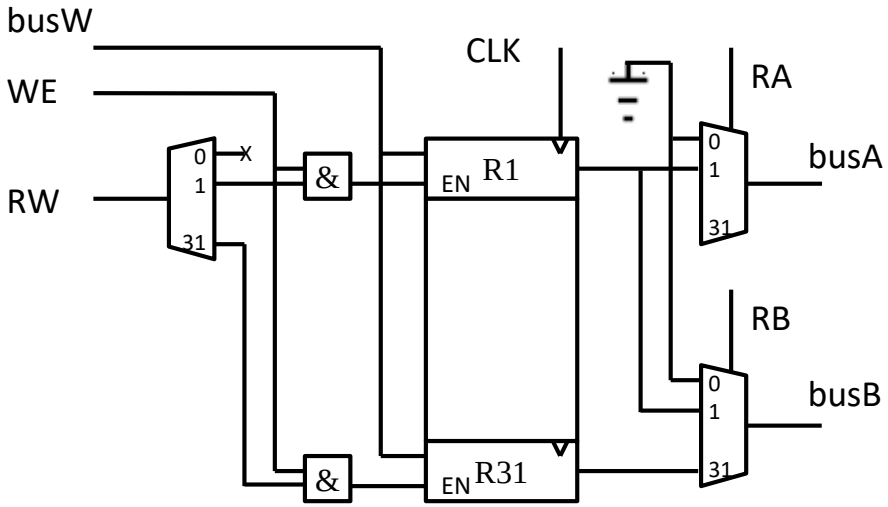
- 采用结构建模方法建模RF
  - 其他部分为行为建模




```
wire         we[31:1] ;
wire [31:0] q[31:1]

genvar i ;
genvar j ;

generate
  for ( i=1; i<32; i=i+1 )
  begin : label_we
    assign we[i] = (RW==i) & WE ;
  end
endgenerate

generate
  for ( j=1; j<32; j=j+1 )
  begin : label_d32
    d32 u_d32(busW, we[j], q[j], clk) ;
  end
endgenerate

assign busA = (RA==0) ? 32'b0 : q[RA] ;
assign busB = …
```

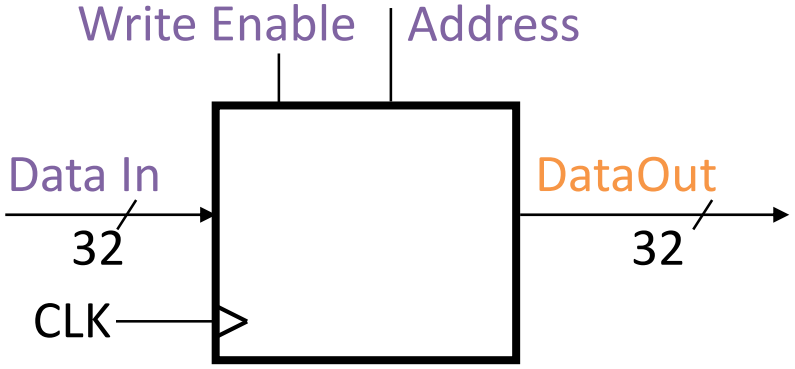
Verilog建模RF

- 用行为建模方法建模RF
- RF写入语句利用了RW是输入信号（即RW是变值）这一特性




```
reg  [31:0] rf[31:1] ;

always @( posedge clk )
    if ( WE )
        rf[RW] <= busW ;

assign busA = (RA==0) ? 32'b0 :
                        rf[RA] ;
assign busB = …
```

Storage Element: Idealized Memory



- Memory (idealized)
  - One input bus: Data In
  - One output bus: Data Out
- Memory access:
  - Read: Write Enable = 0, data at Address is placed on Data Out
  - Write: Write Enable = 1, Data In written to Address
- Clock input (CLK)
  - CLK input is a factor ONLY during write operation
  - During read, behaves as a combinational logic block: Address valid → Data Out valid after "access time"

Verilog建模存储器

- 建模要点：内部是寄存器阵列
- 时序特点：写入的数据滞后1个cycle输出
  - 由寄存器特性决定

```
 1 module MEM4KB( A, DI, We, DO, clk ) ;
 2     input  [9:0]  A;
 3     input  [31:0] DI ;
 4     input         We ;
 5     output [31:0] DO ;
 6     input         clk ;
 7
 8     reg    [31:0] array[1023:0] ;
 9
10     assign DO = array[A] ;
11
12     always @( posedge clk )
13         if ( We )
14             array[A] <= DI ;
15 endmodule
```

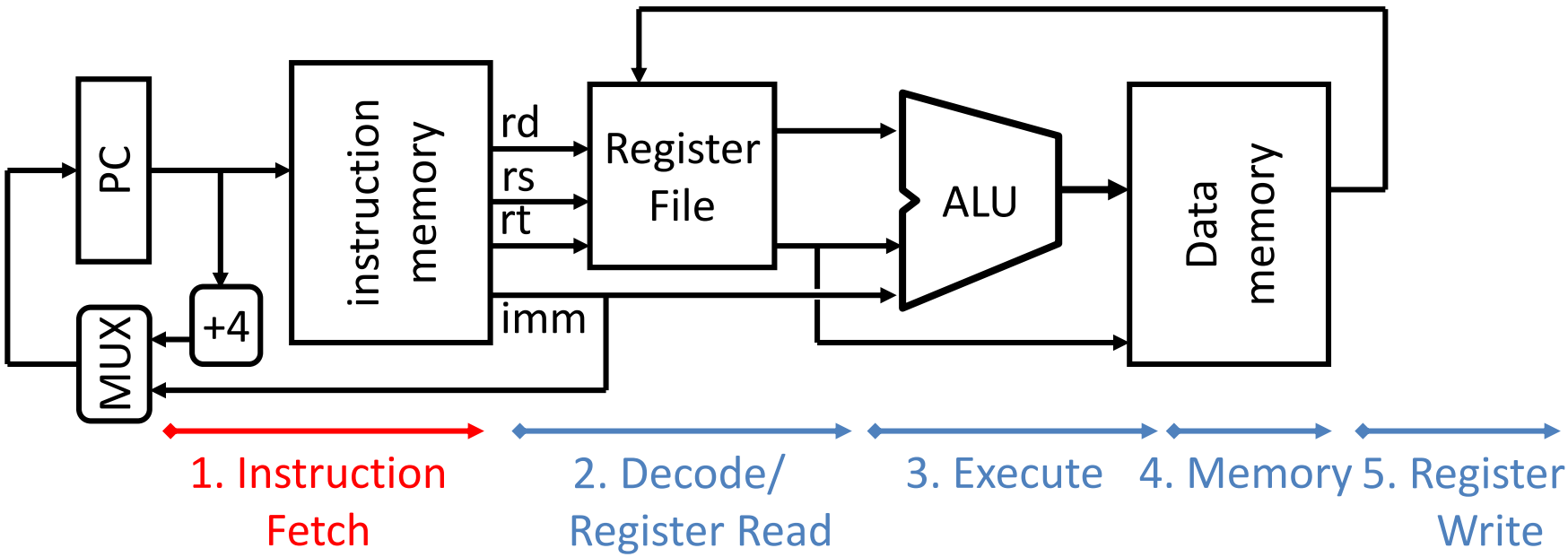
TIP：
建模类似于RF。
实际设计芯片时，会采用定制的库，而不会用寄存器方式实现存储器。

对于P8，存储器要用FPGA芯片内置的块存储器。

提纲

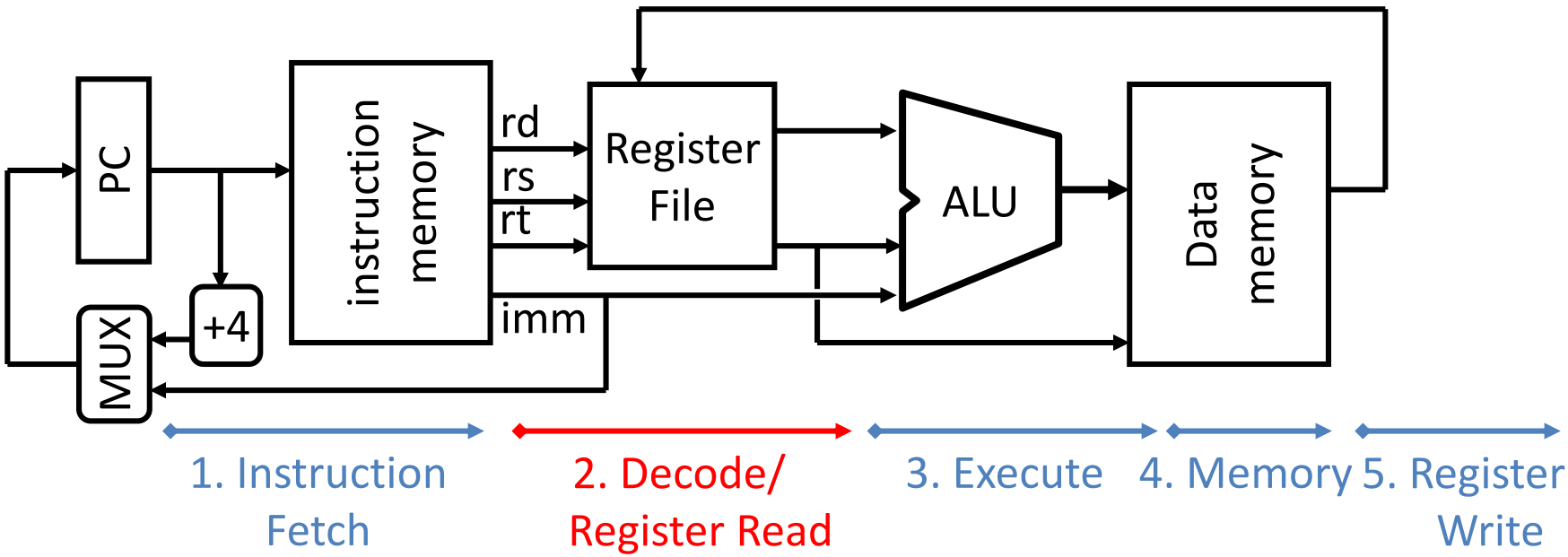
- 内容主要取材
  - CS617的20讲
- 处理器设计
- 数据通路概述
- 组装数据通路
- 控制介绍

Datapath Overview (1/5)



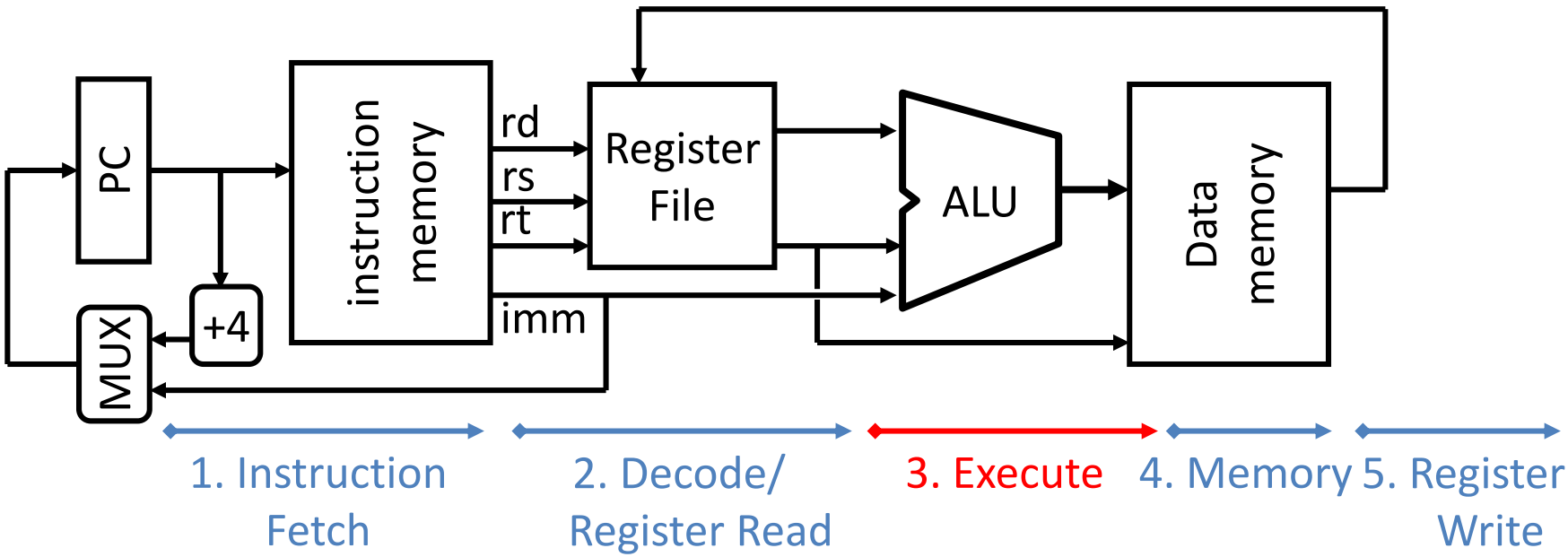
- Phase 1: *Instruction Fetch* (IF)
  - Fetch 32-bit instruction from memory
  - Increment PC (PC = PC + 4)

Datapath Overview (2/5)



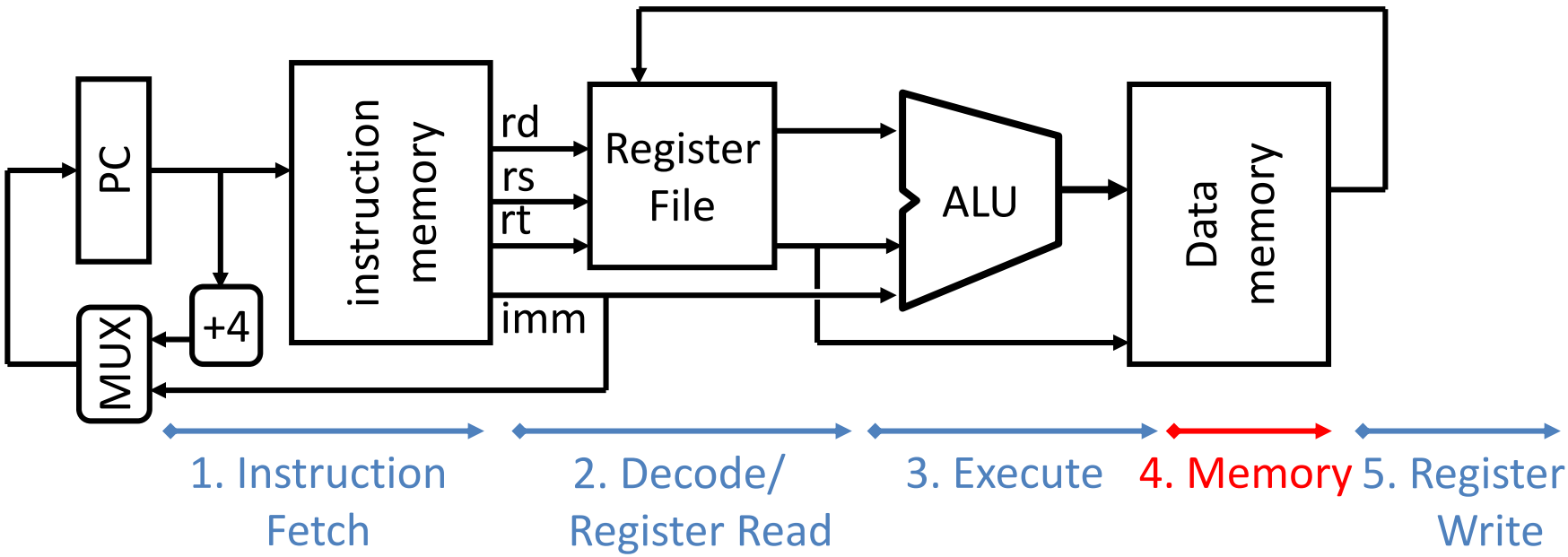
- Phase 2: *Instruction Decode* (ID)
  - Read the opcode and appropriate fields from the instruction
  - Gather all necessary registers values from Register File

Datapath Overview (3/5)



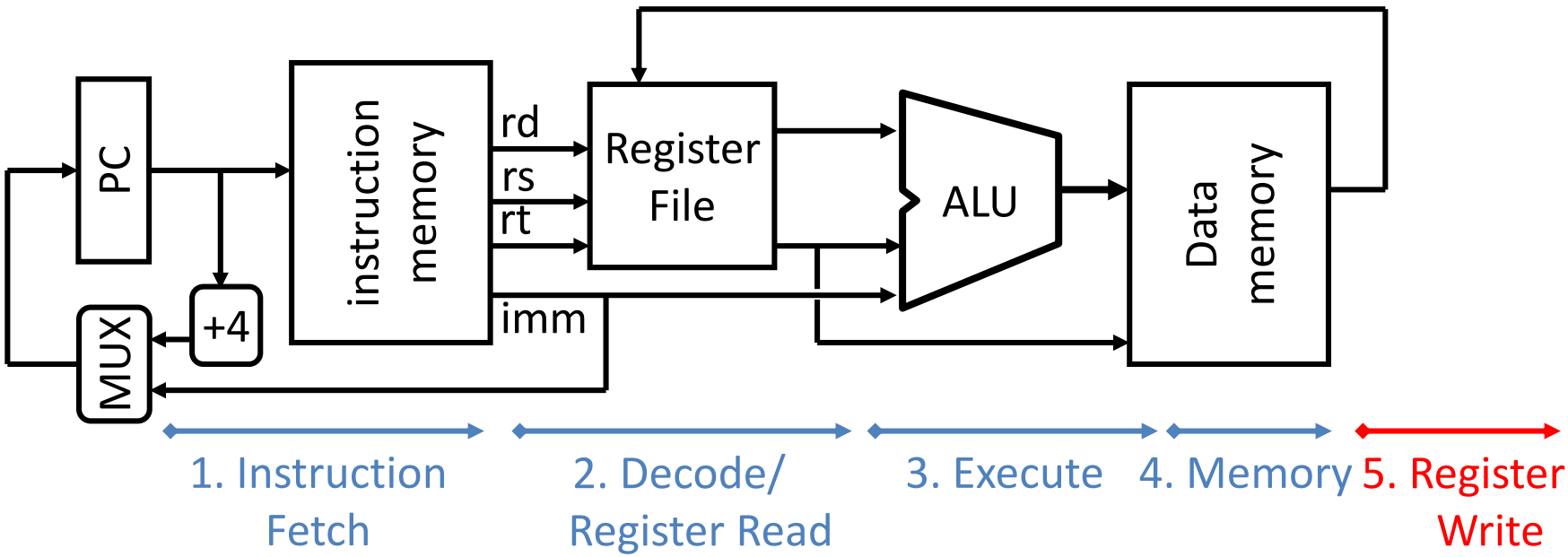
- Phase 3: *Execute* (EX)
  - ALU performs operations: arithmetic (`+`,`-`,`*`,`/`), shifting, logical (`&`,`|`), comparisons (`slt`,`==`)
  - Also calculates addresses for loads and stores

Datapath Overview (4/5)



- Phase 4: *Memory Access* (MEM)
  - Only load and store instructions do anything during this phase; the others remain idle or skip this phase
  - Should be fast due to caches

Datapath Overview (5/5)



- Phase 5: *Register Write* (WB for "write back")
  - Write the instruction result back into the Register File
  - Those that don't (e.g. `sw`, `j`, `beq`) remain idle or skip this phase

Why Five Stages?

- Could we have a different number of stages?
  - Yes, and other architectures do
- So why does MIPS have five if instructions tend to idle for at least one stage?
  - The five stages are the union of all the operations needed by all the instructions
  - There is one instruction that uses all five stages: *load* (`lw`/`lb`)

提纲

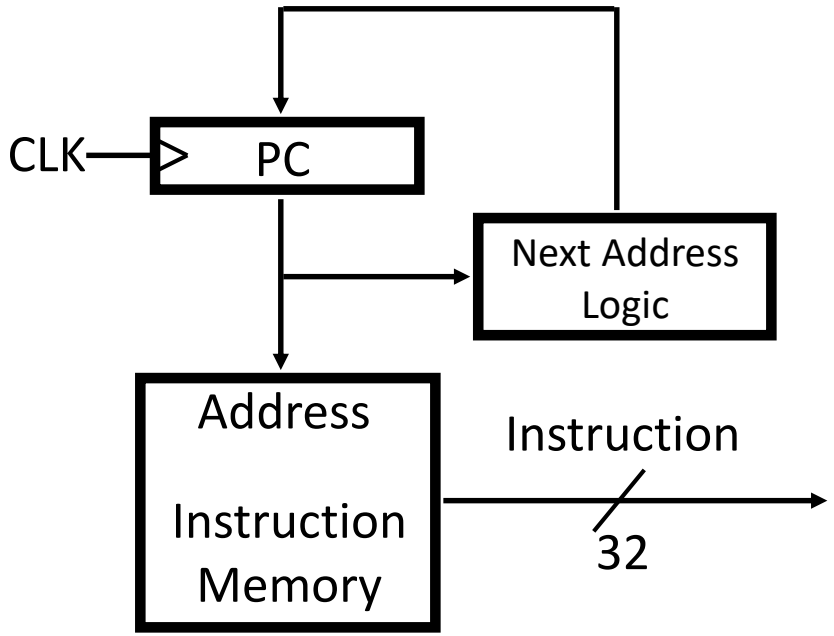
- 内容主要取材
  - CS617的20讲
- 处理器设计
- 数据通路概述
- 组装数据通路
- 控制介绍

Step 3: Assembling the Datapath

- Assemble datapath to meet RTL requirements
  - Exact requirements will change based on ISA
  - Here we will examine *each instruction* of MIPS-lite
- The datapath is all of the hardware components and wiring necessary to carry out all of the different instructions
  - Make sure all components (e.g. RegFile, ALU) have access to all necessary signals and buses
  - Control will make sure instructions are properly executed (the decision making)

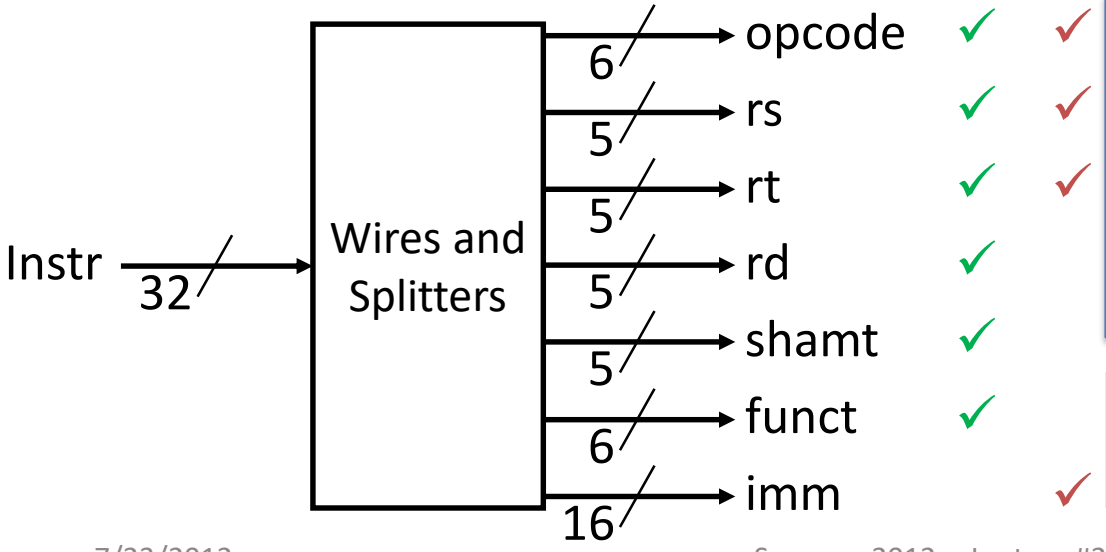
Datapath by Instruction

- All instructions: *Instruction Fetch* (**IF**)
  - Fetch the Instruction: mem[PC]
  - Update the program counter:
    - Sequential Code:
      PC ← PC + 4
    - Branch and Jump:
      PC ← "something else"



Datapath by Instruction

- All instructions: *Instruction Decode* (**ID**)
  - Pull off all relevant fields from instruction to make available to other parts of datapath
    - MIPS-lite only has **R-format** and **I-format**
    - Control will sort out the proper routing (discussed later)

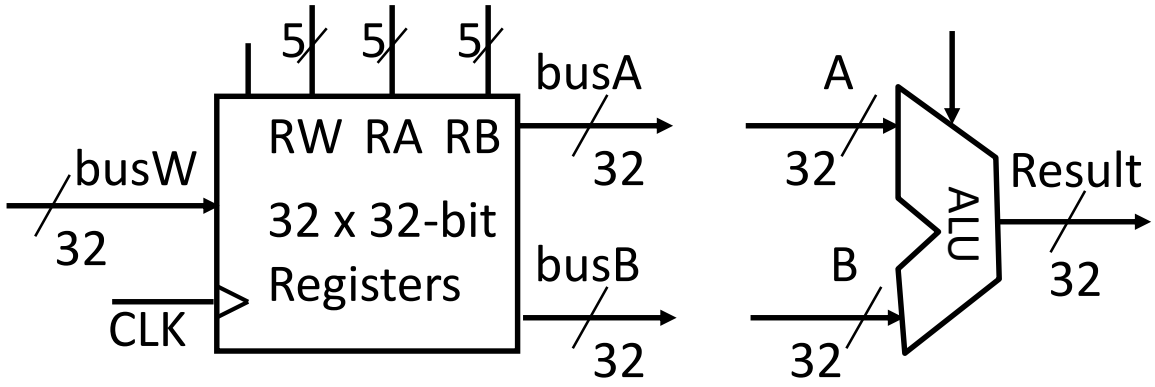


TIP：
原理与一路入户电分成多路室内电相同。
以Instr[5:0]为例，分一路为funct，另一路为imm的最低5位。

```
assign funct = Intr[5:0] ;
assign imm = Instr[15:0]
```

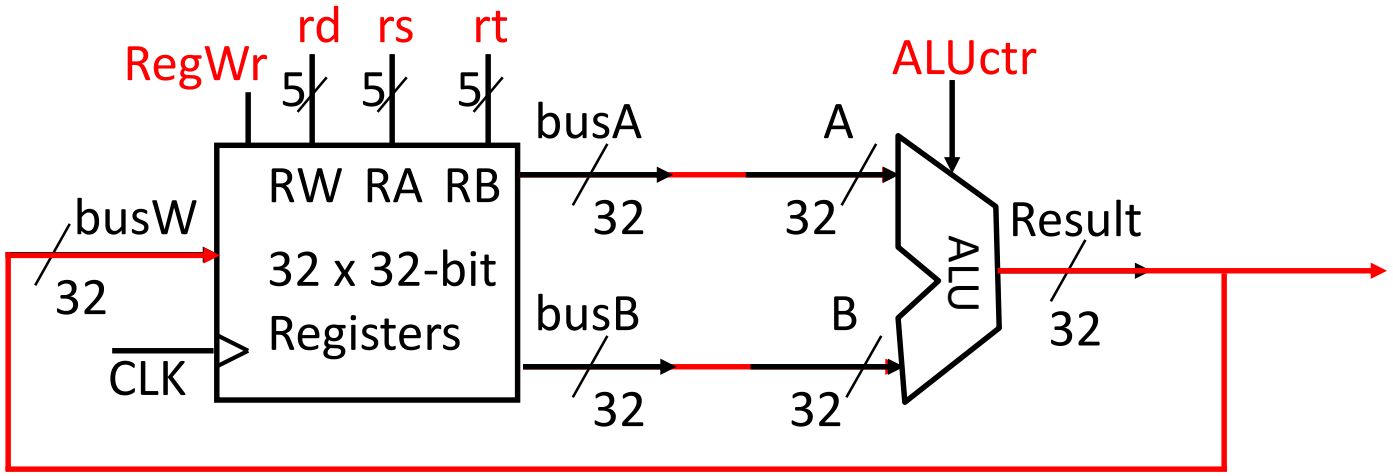
Step 3: Add & Subtract

- `ADDU R[rd]←R[rs]+R[rt];`
- Hardware needed:
  - Instruction Mem and PC (already shown)
  - Register File (RegFile) for read and write
  - ALU for add/subtract



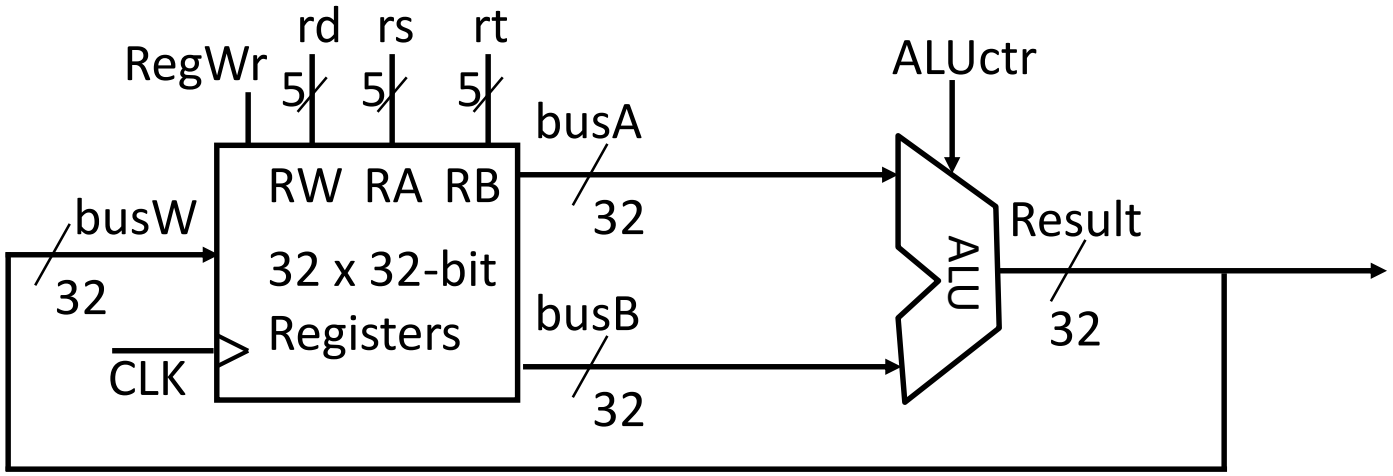
Step 3: Add & Subtract

- `ADDU R[rd]←R[rs]+R[rt];`
- Connections:
  - RegFile and ALU Inputs
  - Connect RegFile and ALU
  - `RegWr` (1) and `ALUctr` (`ADD/SUB`) set by control in **ID**

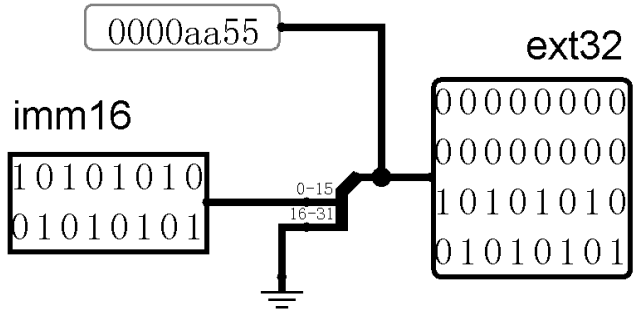


Step 3: Or Immediate

- `ORI R[rt]←R[rs]|zero_ext(Imm16);`
- Is the hardware below sufficient?
  - Zero extend `imm16`?
  - Pass `imm16` to input of ALU?
  - Write result to `rt`?



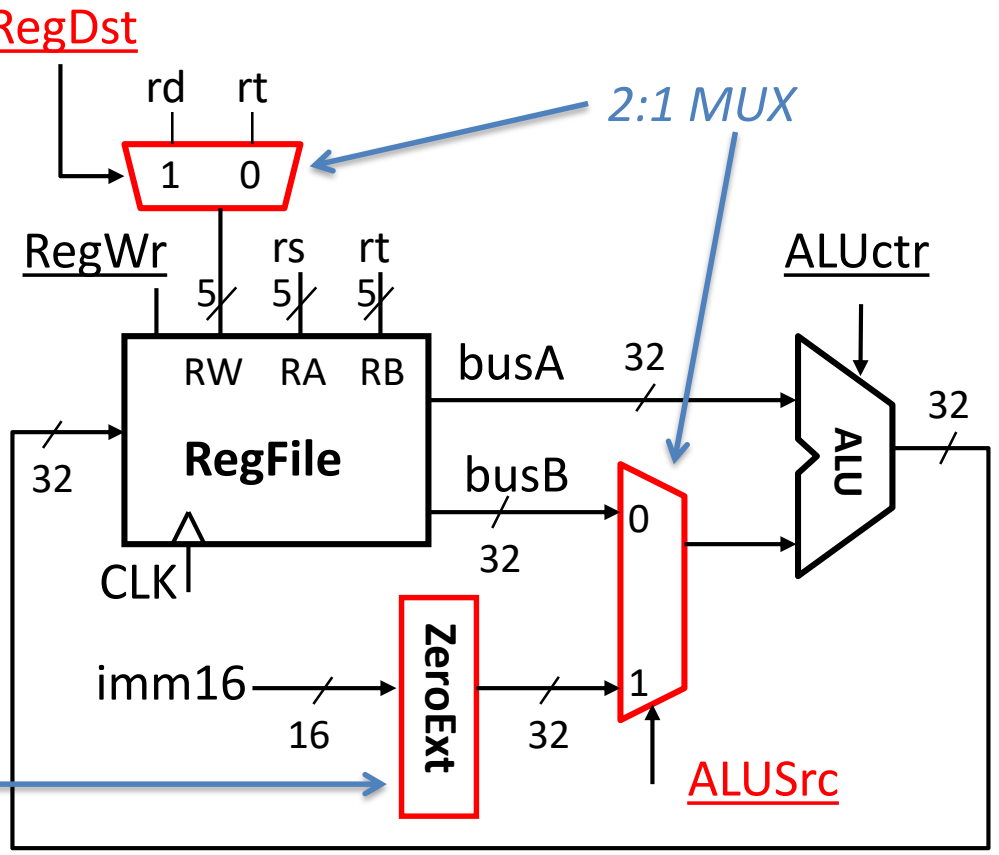
Step 3: Or Immediate



- `ORI R[rt]←R[rs]|zero_ext(Imm16);`
- Add new hardware:
  - Still support add/sub
  - New control signals:
    `RegDst, ALUSrc`

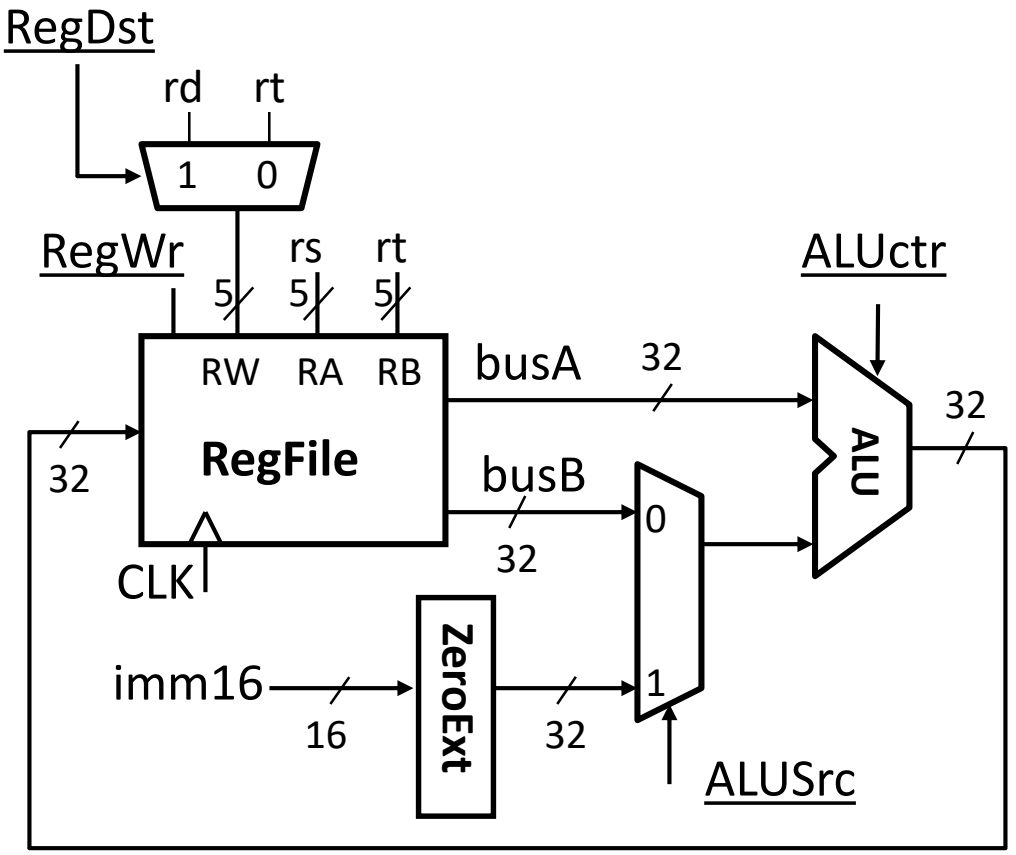
```
input  [15:0] imm16 ;
output [31:0] ext32 ;

assign ext32 = {16'b0, imm16}
```

How to implement this?



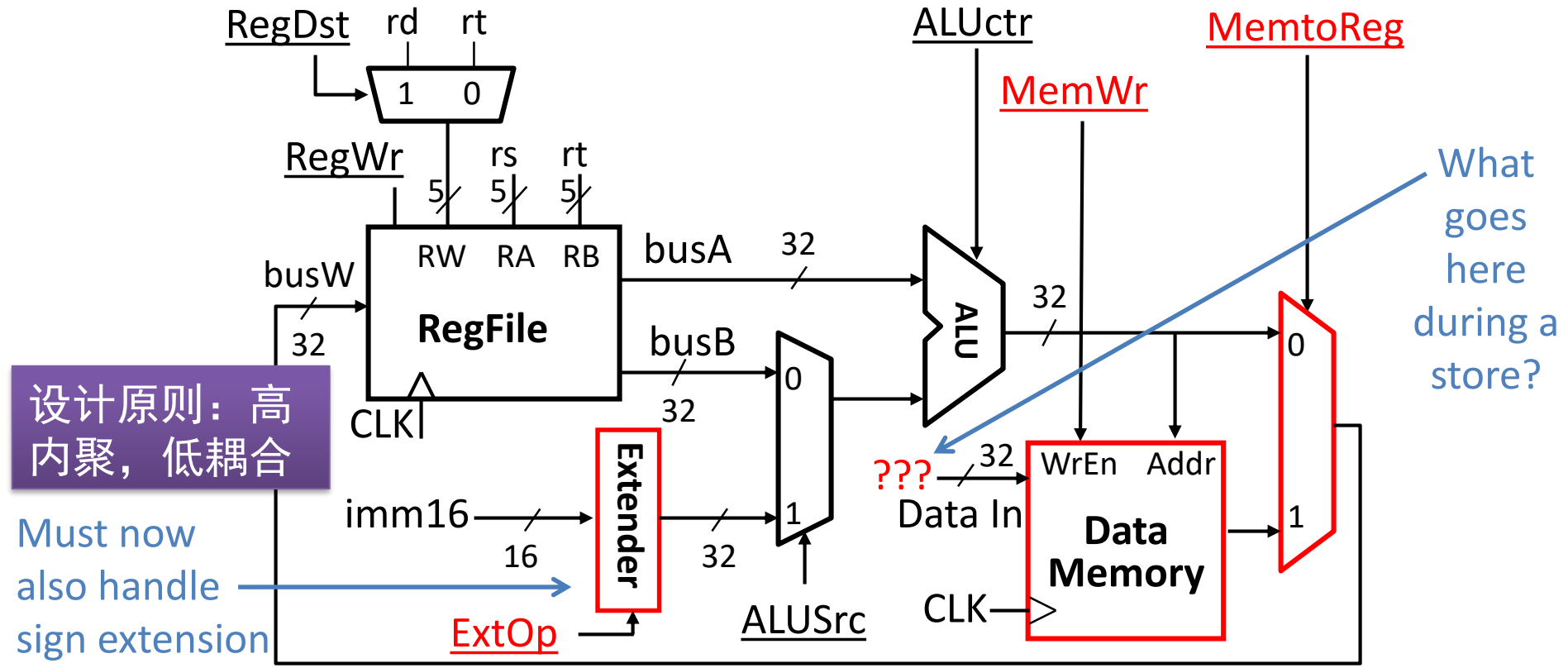
Step 3: Load

- `LOAD R[rt]←MEM[R[rs]+sign_ext(Imm16)];`
- Hardware sufficient?
  - Sign extend `imm16`?
  - Where's MEM?



Step 3: Load

- `LOAD R[rt]←MEM[R[rs]+sign_ext(Imm16)];`
- New control signals: `ExtOp`, `MemWr`, `MemtoReg`



Verilog建模扩展单元

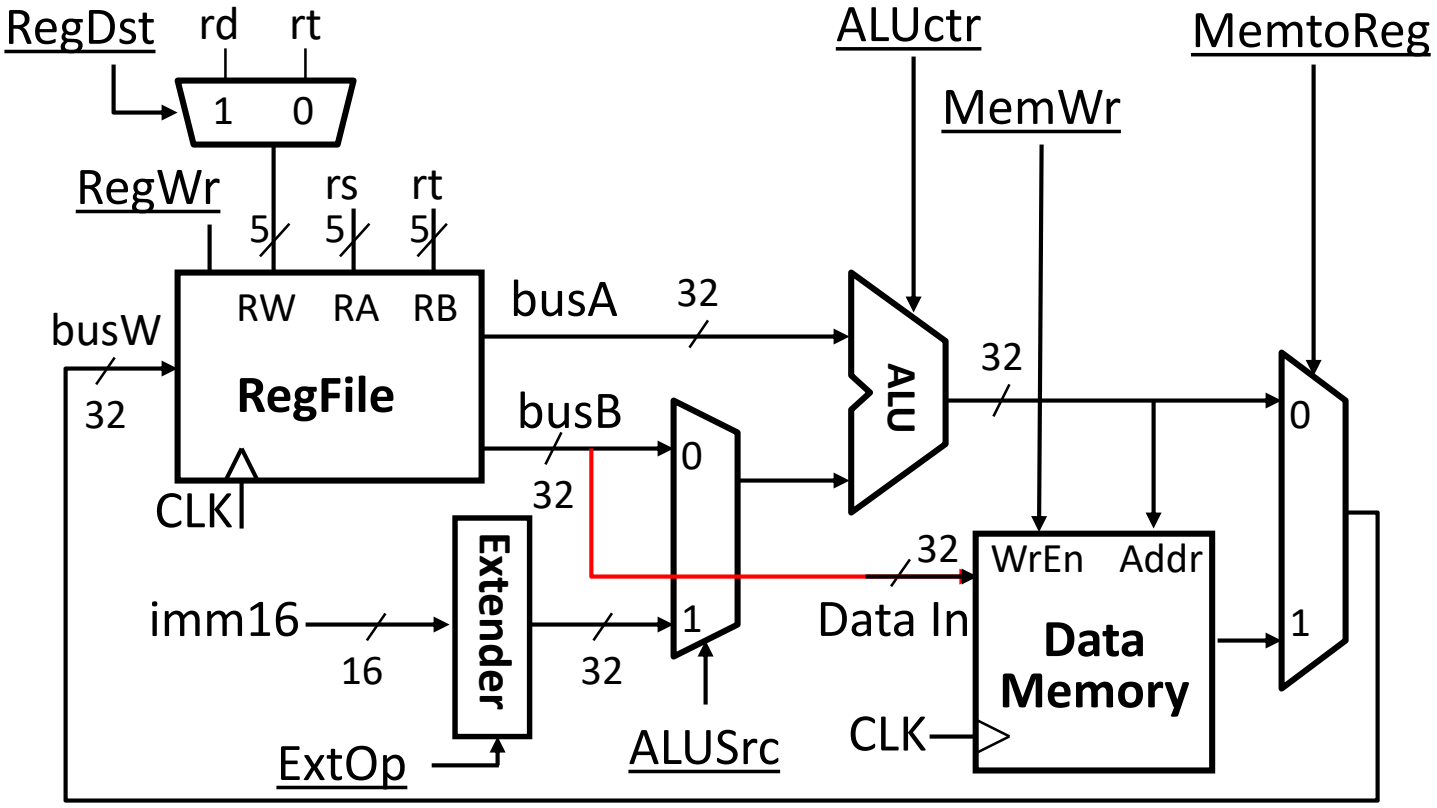
- 建模要点：二进制补码的扩展方法

```
1  module EXT32( imm16, extop, ext32 ) ;
2      input  [15:0] imm16;
3      input         extop ;
4      output [31:0] ext32 ;
5
6      assign ext32 = extop ? {16{imm16[15]}, imm16} :
7                             {16'b0, imm16} ;
8
9  endmodule
```

Q:
如何改善行6的编码风格以使其可读性更好？
A:
哪怕extop的选项很少，也应该用宏来定义extop的各个取值。增加可读性是降低代码开发与调试复杂度的重要手段。

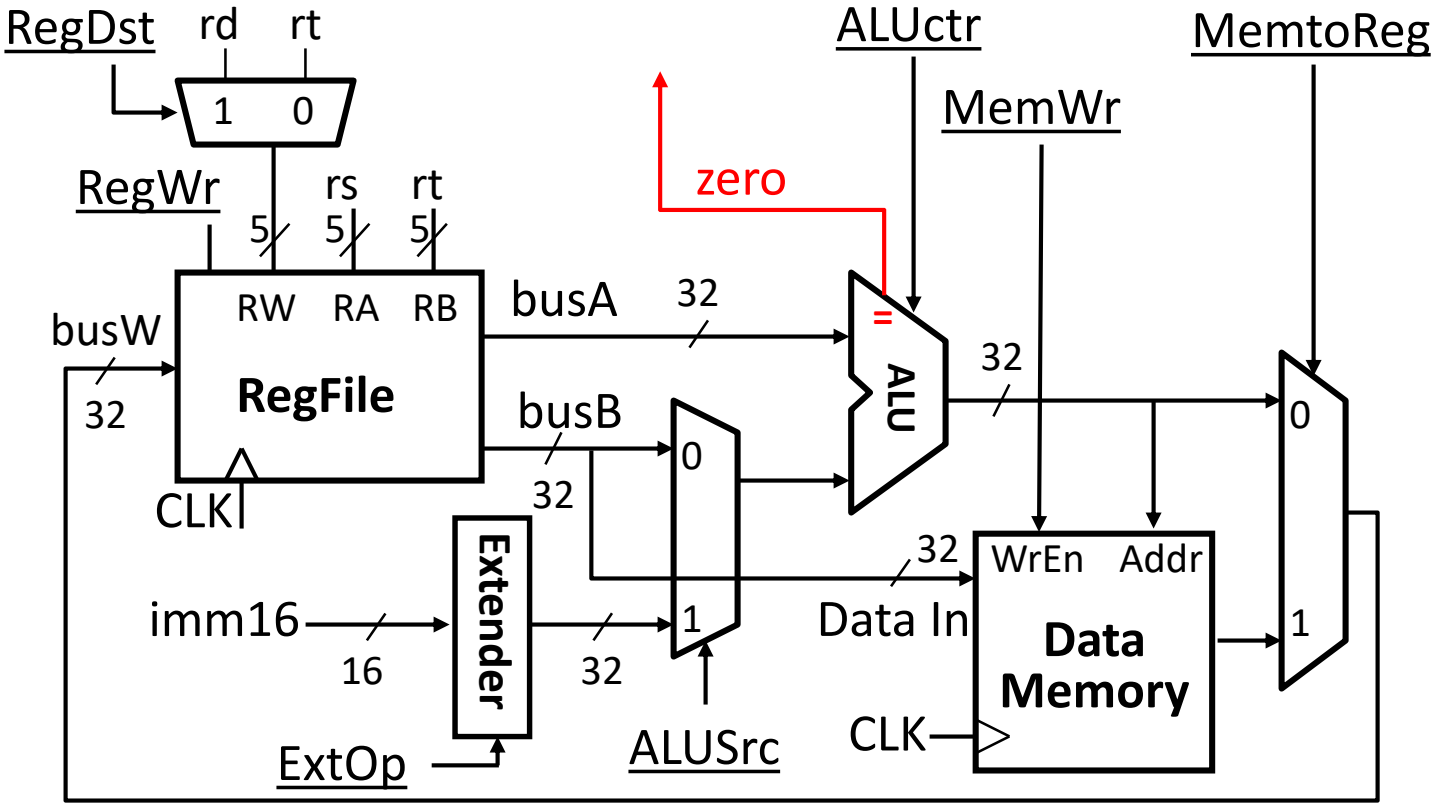
Step 3: Store

- `STORE MEM[R[rs]+sign_ext(Imm16)]←R[rt];`
- Connect `busB` to `Data In` (no extra control needed!)



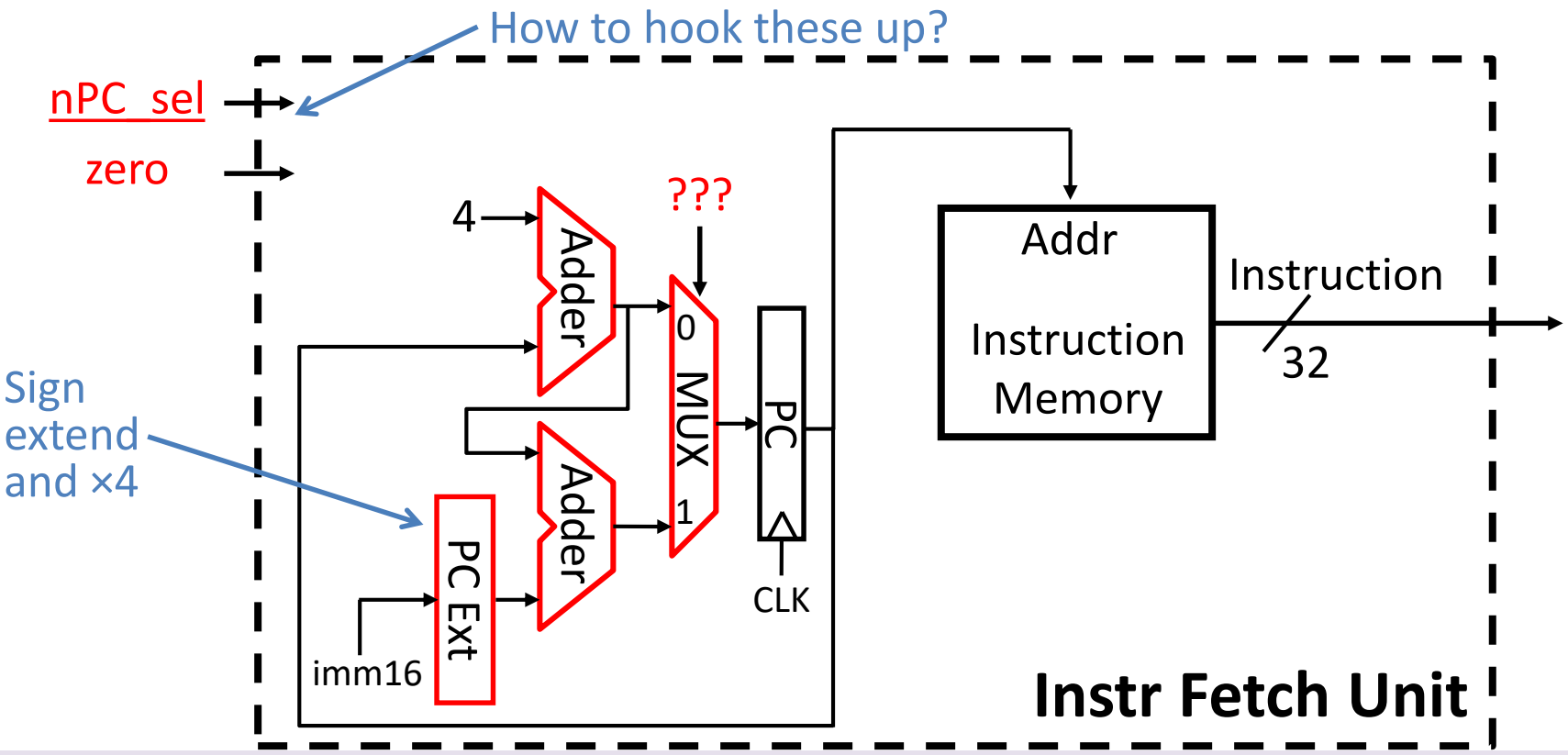
Step 3: Branch If Equal

- BEQ if(R[rs]==R[rt]) then PC←PC+4 + (sign_ext(Imm16) || 00)
- Need comparison output from ALU



Step 3: Branch If Equal

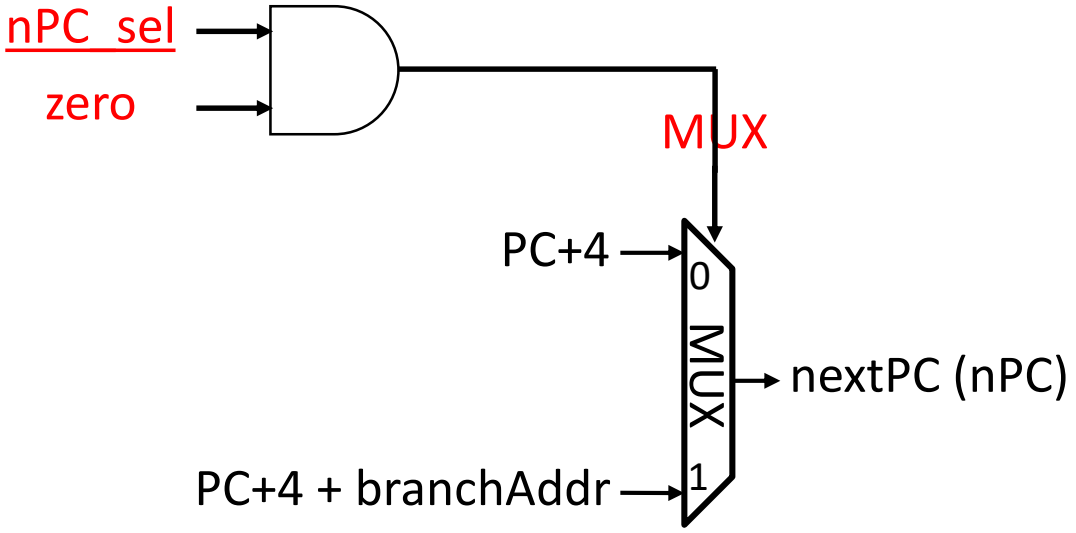
- BEQ if(R[rs]==R[rt]) then PC←PC+4 + (sign_ext(Imm16) || 00)
- Revisit "next address logic":



注意：硬件的计算顺序与软件的计算顺序有本质不同。并行是核心性质！

Step 3: Branch If Equal

- BEQ if(R[rs]==R[rt]) then PC←PC+4 + (sign_ext(Imm16) || 00)
- Revisit "next address logic":
  - nPC_sel should be 1 if branch, 0 otherwise

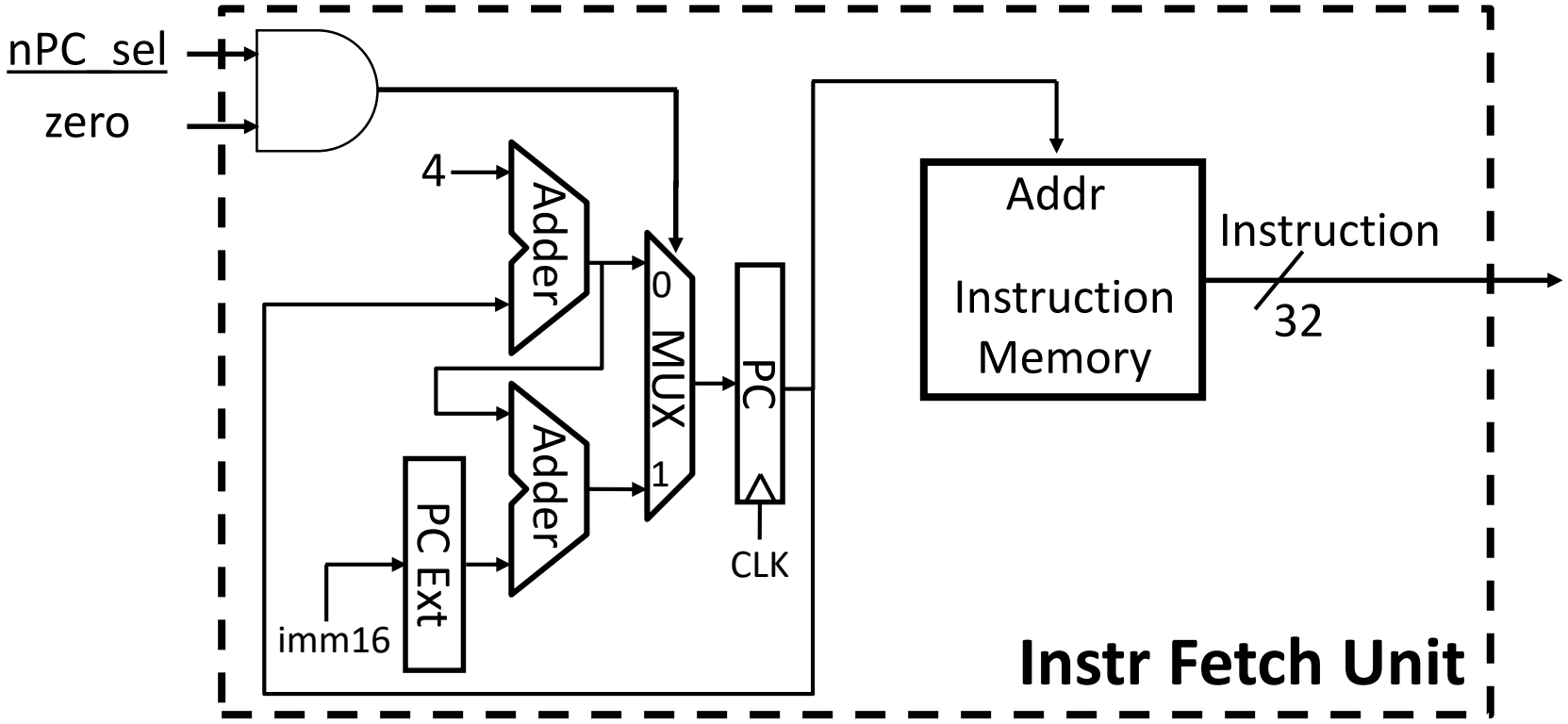


nPC_sel	zero	MUX
0	0	0
0	1	0
1	0	0
1	1	1

How does this change if we add `bne`*?*

Step 3: Branch If Equal

- BEQ if(R[rs]==R[rt]) then PC←PC+4 + (sign_ext(Imm16) || 00)
- Revisit "next address logic":

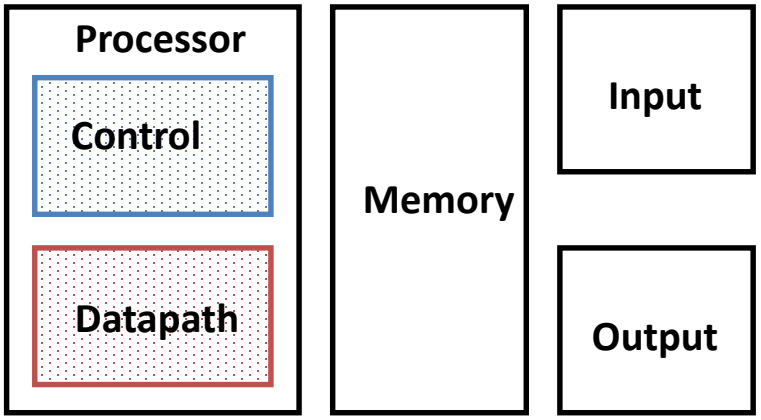


提纲

- 内容主要取材
  - CS617的20讲
- 处理器设计
- 数据通路概述
- 组装数据通路
- 控制介绍

Processor Design Process

- Five steps to design a processor:
  1. Analyze instruction set → datapath requirements
  2. Select set of datapath components & establish clock methodology
  3. Assemble datapath meeting the requirements
  4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer
  5. Assemble the control logic
     - Formulate Logic Equations
     - Design Circuits



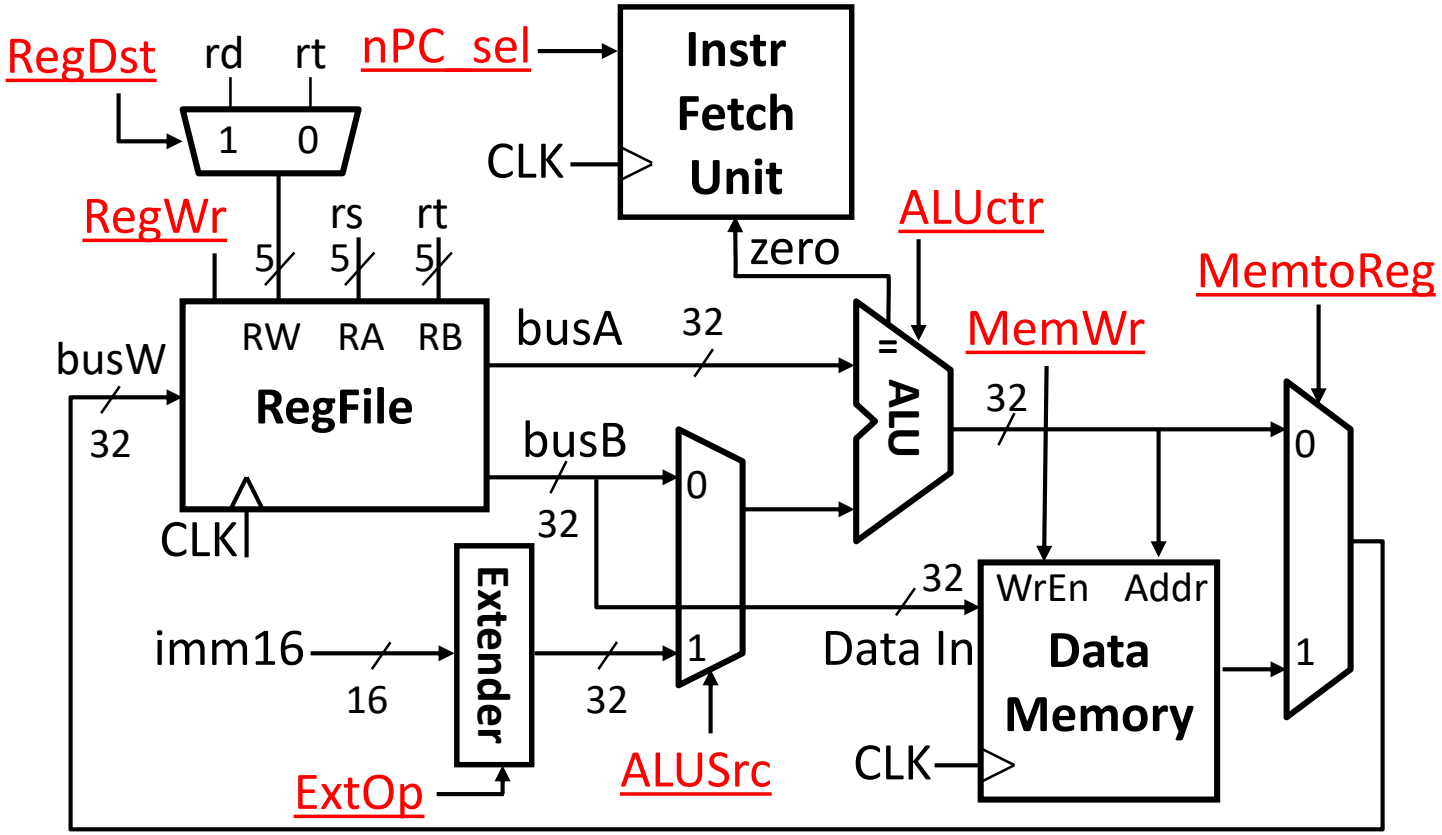
Now

Control

- Need to make sure that correct parts of the datapath are being used for each instruction
  - Have seen *control signals* in datapath used to select inputs and operations
  - For now, focus on what value each control signal should be for each instruction in the ISA
  - Next lecture, we will see how to implement the proper combinational logic to implement the control

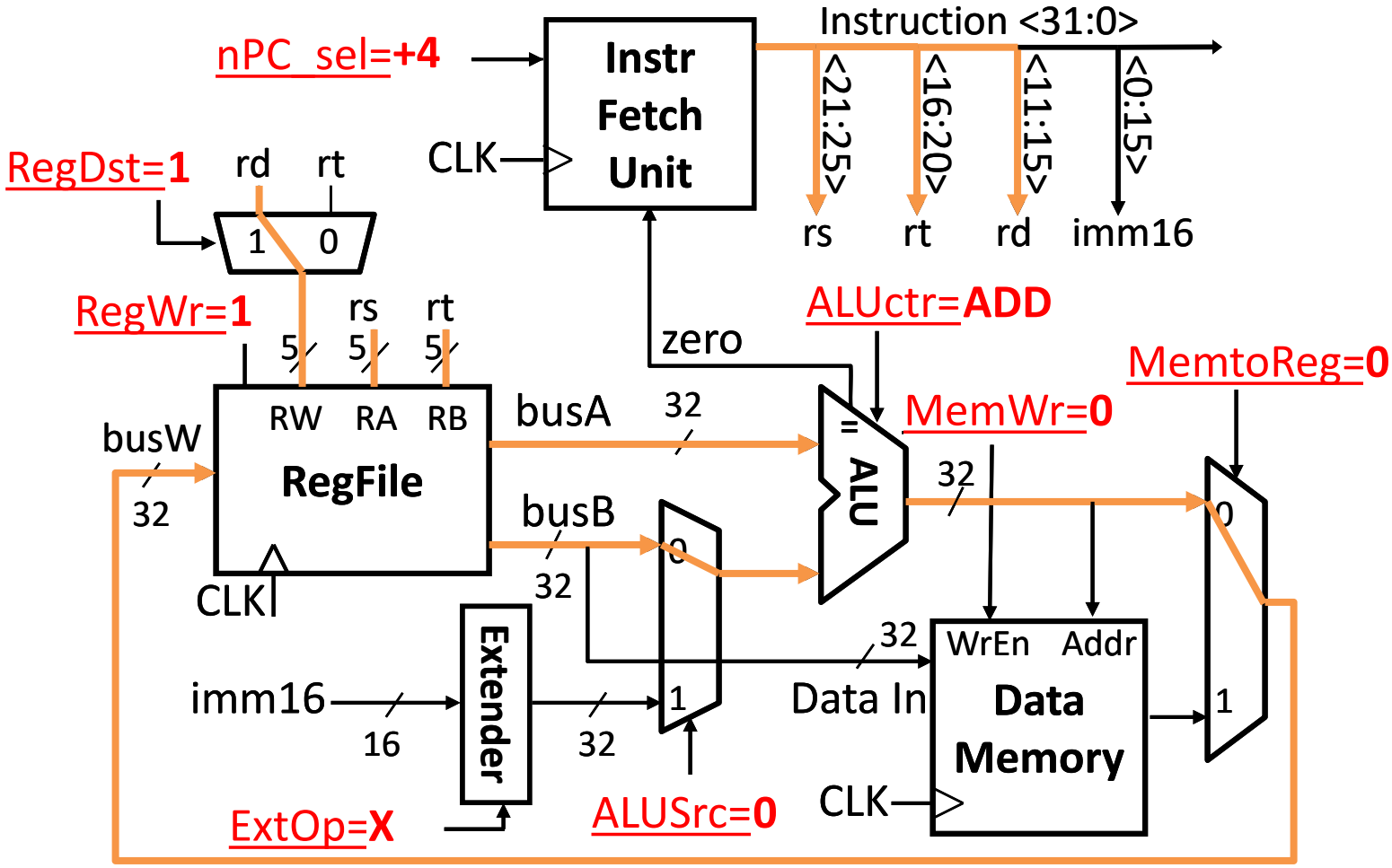
MIPS-lite Datapath Control Signals

- **ExtOp:** 0 → "zero"; 1 → "sign"
- **ALUsrc:** 0 → busB; 1 → imm16
- **ALUctr:** "ADD", "SUB", "OR"
- **nPC_sel:** 0 → +4; 1 → branch
- **MemWr:** 1 → write memory
- **MemtoReg:** 0 → ALU; 1 → Mem
- **RegDst:** 0 → "rt"; 1 → "rd"
- **RegWr:** 1 → write register



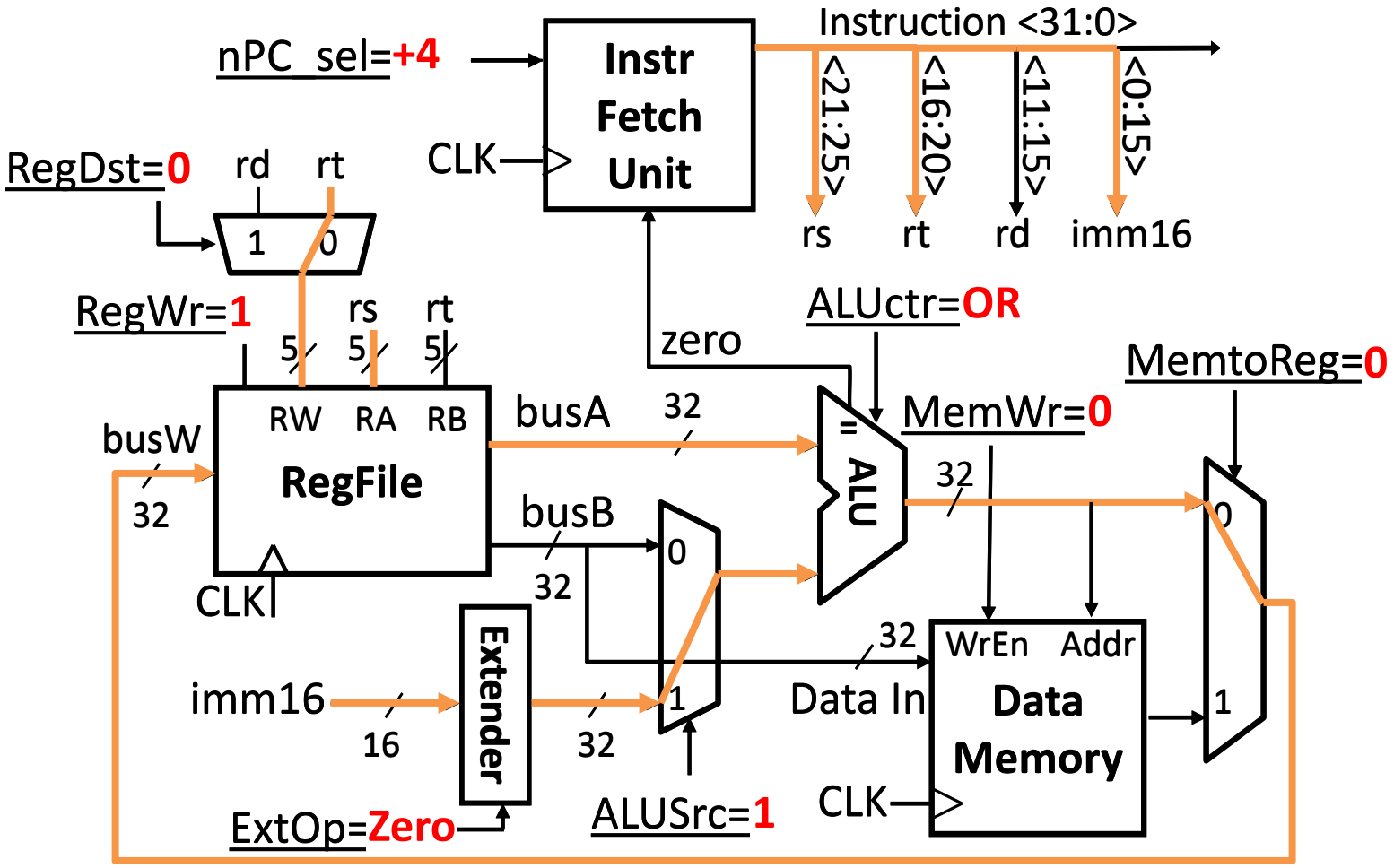
Desired Datapath For addu

- R[rd]←R[rs]+R[rt];



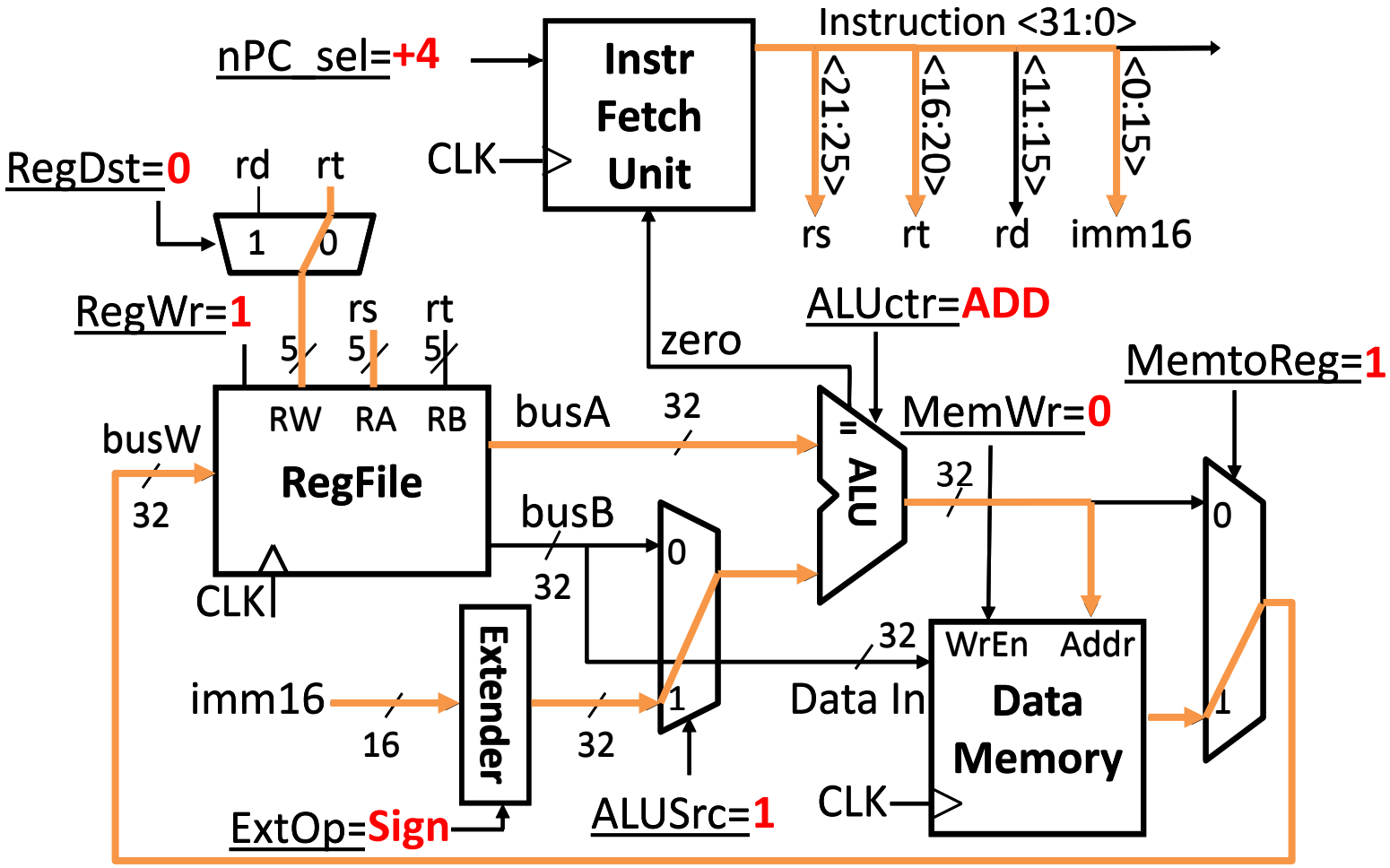
Desired Datapath For `ori`

- `R[rt]←R[rs]|ZeroExt(imm16);`



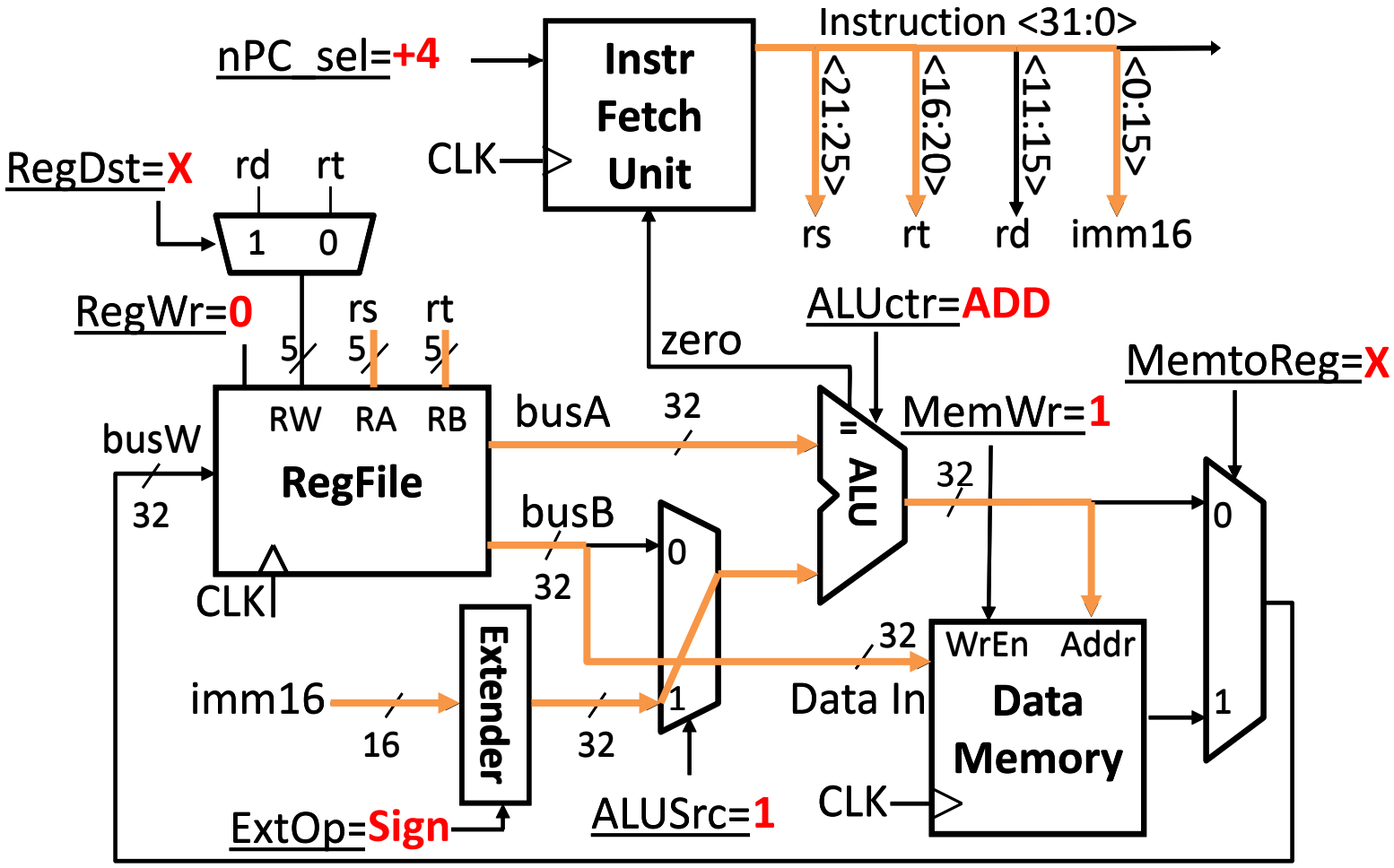
Desired Datapath For `load`

- `R[rt]←MEM{R[rs]+SignExt[imm16]};`



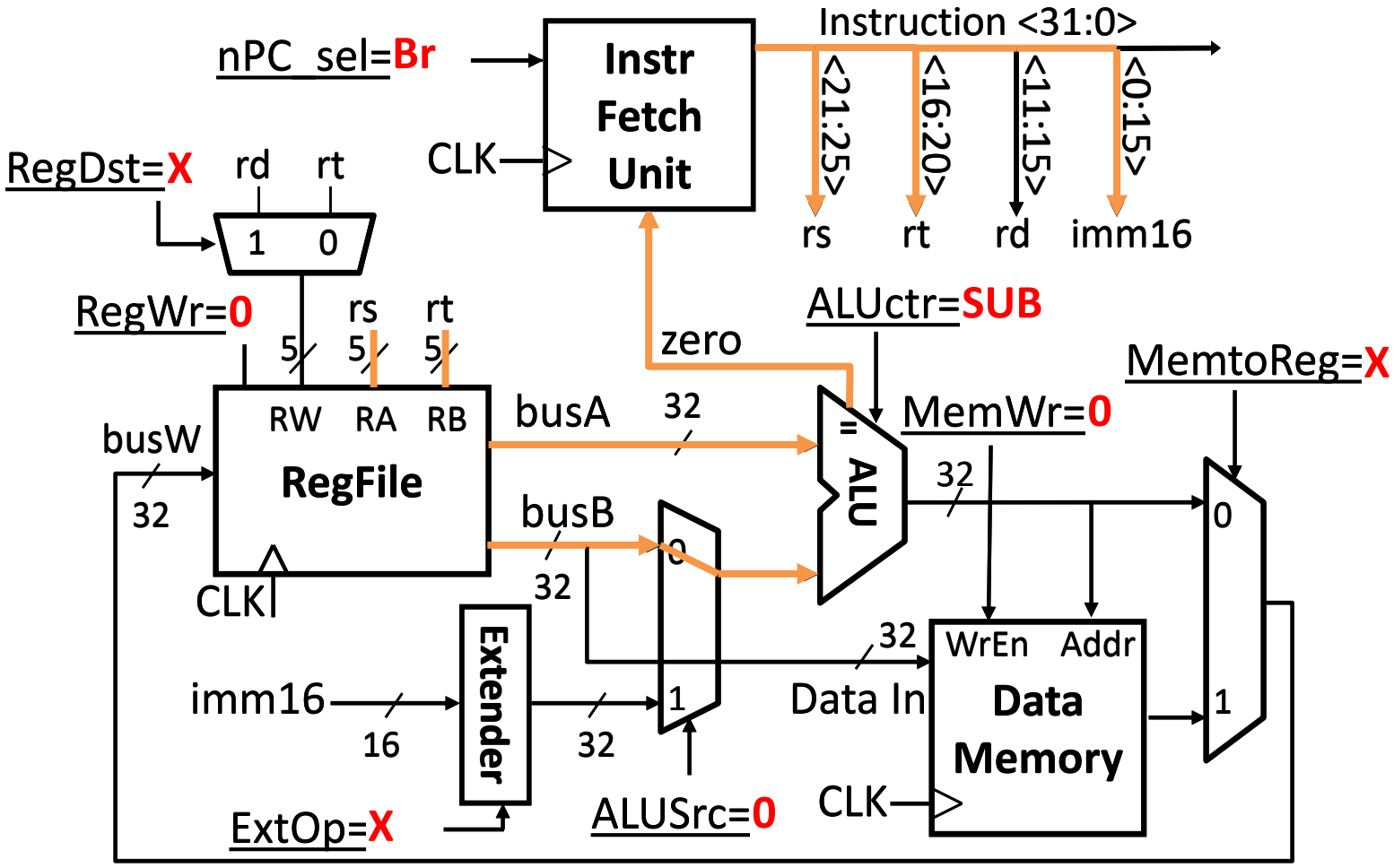
Desired Datapath For `store`

- `MEM{R[rs]+SignExt[imm16]}←R[rt];`



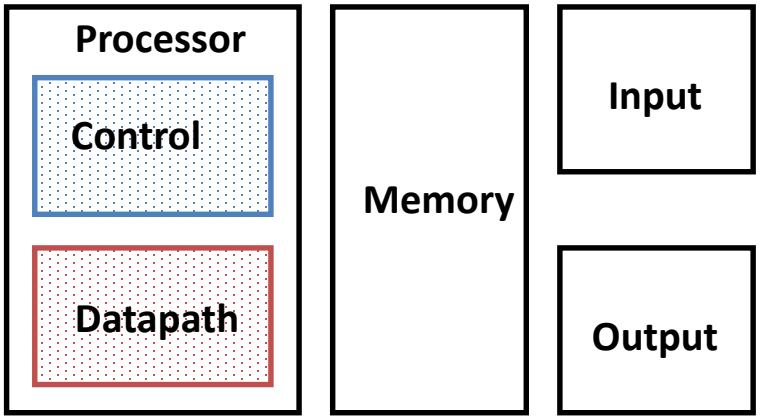
Desired Datapath For beq

- BEQ if(R[rs]==R[rt]) then PC←PC+4 + (sign_ext(Imm16) || 00)



Summary (1/2)

- Five steps to design a processor:
  1) Analyze instruction set → datapath requirements
  2) Select set of datapath components & establish clock methodology
  3) Assemble datapath meeting the requirements
  4) Analyze implementation of each instruction to determine setting of control points that effects the register transfer
  5) Assemble the control logic
     - Formulate Logic Equations
     - Design Circuits



Summary (2/2)

- Determining control signals
  - Any time a datapath element has an input that changes behavior, it requires a control signal (e.g. ALU operation, read/write)
  - Any time you need to pass a different input based on the instruction, add a MUX with a control signal as the selector (e.g. next PC, ALU input, register to write to)
- Your datapath and control signals will change based on your ISA